AFRL-IF-RS-TR-2003-62
Final Technical Report
March 2003

COMPUTATIONAL VIDEO FOR COLLABORATIVE APPLICATIONS

Massachusetts Institute of Technology

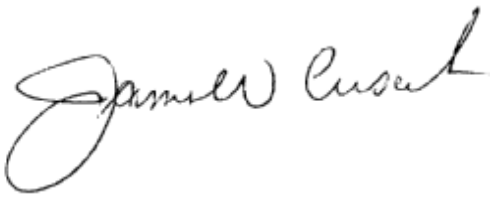
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-62 has been reviewed and is approved for publication.

APPROVED: 

EDWARD DEPALMA
Project Engineer

FOR THE DIRECTOR: 

JAMES W. CUSACK, Chief
Information Systems Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MARCH 2003	Final Apr 97 – Aug 02

4. TITLE AND SUBTITLE
COMPUTATIONAL VIDEO FOR COLLABORATIVE APPLICATIONS

5. FUNDING NUMBERS
C - F30602-97-1-0283
PE - 62301E
PR - F305
TA - 01
WU - 00

6. AUTHOR(S)
Leonard McMillan and David Gifford

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology
Laboratory for Computer Science
77 Massachusetts Avenue, E19-719
Cambridge Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFSF
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-62

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Edward DePalma/IFSF/(315) 330-3069/ Edward.DePalma@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
The Computational Video for Collaborative Applications effort was devoted to the research of video compression/decompression, virtual telepresence, and real-time processing of streaming data-types. Compression techniques were focused on overcoming the problem of delivering streaming content over unreliable packet-switching networks. Virtual telepresence work was in the development of video streaming with the end user having more control over how the video is presented. Real-time processing of streaming data types involves the fusing together of multiple video streams. A derivative of this work is the development of a new method for constructing virtual camera views from multiple live video streams by using dynamically reparameterized light fields (DRLFs). This technique has the advantage of providing visualizations of a remote scene's background and foreground objects. Extensive involvement at the Siggraph computer graphics conventions over the past several years has furthered the reach and applications of this important research.

14. SUBJECT TERMS
Video Compression, Video Decompression, Video Streaming, Virtual Telepresence, Dynamically Reparameterized Light Fields, Data Fusing

15. NUMBER OF PAGES
90

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Table of Figures

Objective:

In this program, we have developed a wide range of new technologies for filtering, processing, and interpreting streaming media types. We refer to these various technology components as Computational Video systems. The key applications where we have applied our computational video systems are in supporting collaboration and situational awareness. The research outcomes of this program include significant new approaches to video compression and decompression, remote telepresence, multiple-sensor fusion, and the real-time processing and filtering of streaming data types.

Approach:

In the Computation Video project we have conducted research in three areas related to the use of streaming real-time media in collaborative applications. These research areas include:

- Video compression and decompression
- Virtual telepresence for collaboration
- Real-time processing of streaming data-types

Our work in video compression and decompression has focused on the problem of delivering streaming content over unreliable packet-switched networks and the support of heterogeneous environments. This is in contrast to previous video-compression standards such as MPEG and H261, where the error free and sequential transmission is assumed. In this program, we developed a new video compression algorithm called NAÏVE, (Network Aware Internet Video Encoding). NAÏVE is resilient to drop outs (packet loss), out of order transmission, and it allows for a wide-range in performance for client platforms.

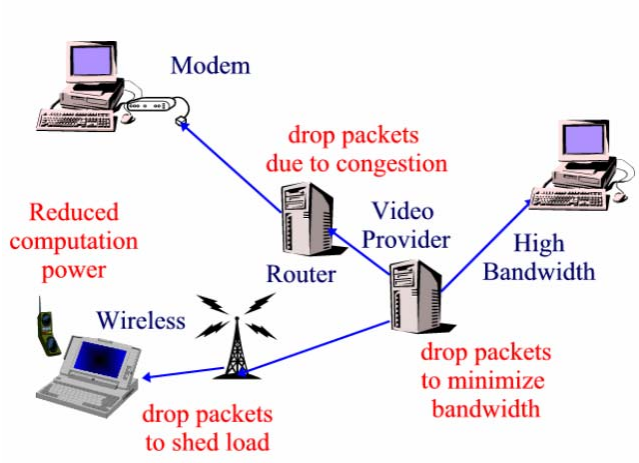


Figure 1. Our Network Aware Internet Video Encoding (NAÏVE) system. Enables the transmission of a single video stream to multiple recipients, without throttling or adapting at the source. It is able to reconstruct video frames in the presence of packet loss or conjestion.

In order to support video in heterogeneous environments we have also developed software-only MPEG decoder written entirely in Java. This package allows for distribution of streaming video content across any Java-enabled platform without requiring any specialized plug-ins or classes. The code can be embedded as part of a web page and then it can be delivered on demand to the client. The entire decoder, including a customizable user interface, is less than 85 Kbytes.

In support of collaboration, we have developed several technologies related to virtual telepresence. By virtual telepresence we refer to the use of client-side computation to extend video conferencing. Our goal was to provide the end-user, or consumer, of a video stream with more control over how that video stream is presented. We sought to provide the end-user with control over a remote virtual camera. Moreover, we can provide this level of control simultaneously to multiple viewers, who are observing common video streams. We completed three research projects in this area.

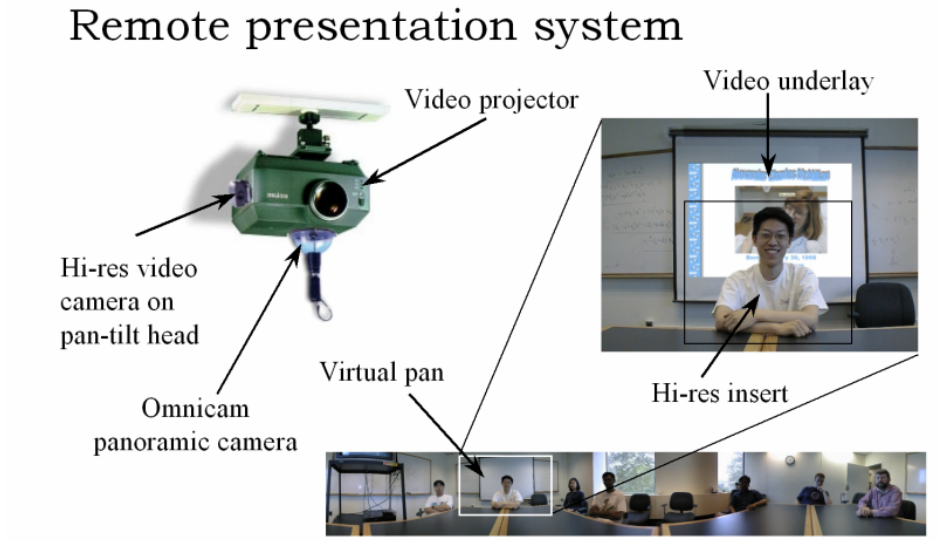


Figure 2. Our remote presentation system combined multiple asynchronous video streams of varying resolutions into a single coherent and navigable media. The end user is provide with pan-tilt and zoom control. Regions of interest are presented at higher resolutions.

We have developed a remote presentation viewing systems, where multiple users are simultaneously allowed to pan, tilt, and change the field-of-view of a virtual camera located at the remote source. This is accomplished by broadcasting a single panoramic video stream. Each viewer customizes his or her viewing experience by controlling a virtual camera. The video image from this virtual camera is synthesized locally at the client. We extended this system to include multiple video streams to allow users to move about even more freely within the remote scene. Auxiliary streams can used to provide increased resolution in areas of interest (i.e. higher resolution of the speaker, or to provide more resolution so that a Powerpoint presentation would be readable). As part of this work we developed techniques for extracting foreground and background objects in real time. These we used to underlay high-resolution display information within a lower resolution panorama.

One unexpected result form our remote presentation viewing system work was a new technology for interpreting and fusing together multiple video streams. We have developed an innovative technology for acquiring three-dimensional models from simultaneous video streams of the same scene, called the image-based visual hull. These three-dimensional models can be synthesized in real-time at rates exceeding 30 frames per second. These models can be used to support many applications. They can be used to provide realistic animated avatars and other content for simulation applications, as well as in intelligence collection for the recognition and classification of targets. Our system can also be used in surveillance applications to provide novel viewpoints other than those seen from any particular camera. Finally, these models represent a compact form of three-dimensional compression, which can be applied to future applications of 3-D teleconferencing.

Another derivative of our remote presentation system was a new method for constructing virtual camera views directly from multiple live video streams, called dynamically re-parameterized light fields. In contrast to our visual hull system, dynamically re-parameterized light fields (DRLFs) have the following advantage, they can provide visualizations of the remote scene's background as well as the foreground objects. In DRLFs the virtual camera's view is interpolated directly from the video stream without constructing an intermediate geometric model. This technique requires a large number of cameras, but it is very efficient computationally. We have demonstrated light field viewers operating at 30 frames per second from 64 live video streams. This technique can also be used to synthesize stereo views.

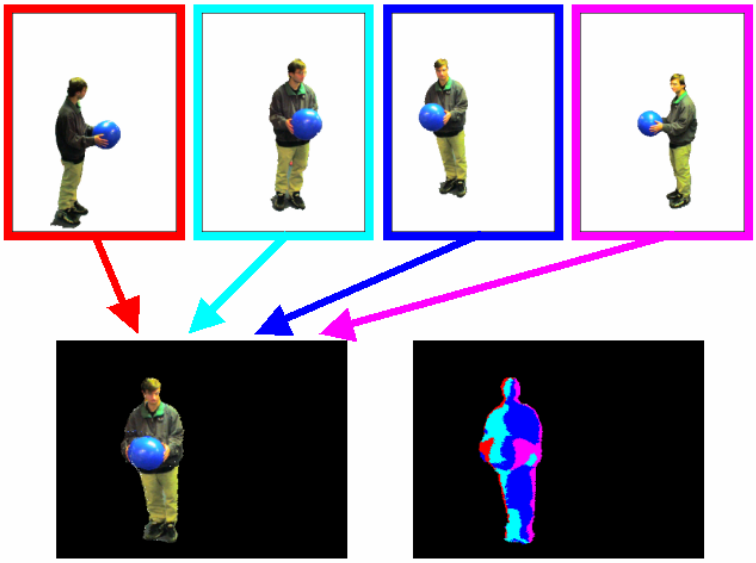


Figure 3. An example output from our image-based visual hull system. Foreground objects from four source cameras are shown above. The lower left image is synthesized from the textures and silhouettes of the source image. The false colored image on the right highlights the contributions of the source images to the final rendered result. Our image-based visual hull system creates models at greater than 10 frames per second. These models can be rendered at speed exceeding 30 frames per second.

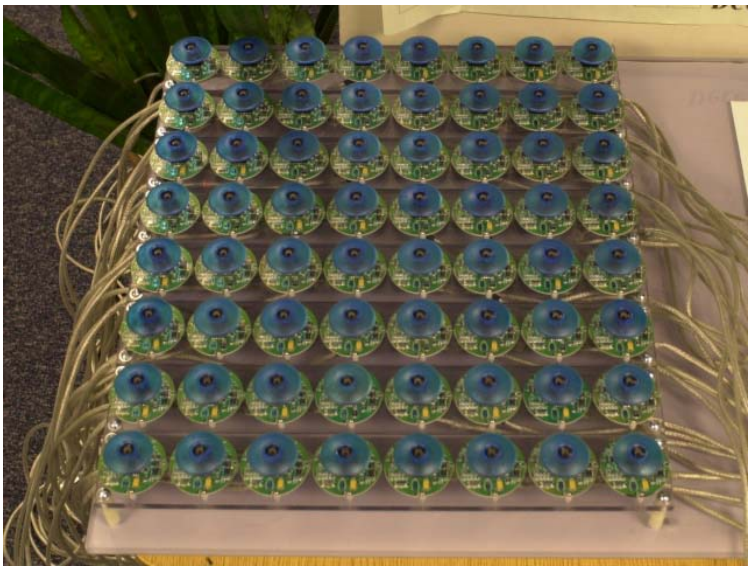


Figure 4. Shown above is a photograph of the 64-camera dynamically re-parameterized light field array. Virtual camera views can be synthesized in real-time using our light field rendering methods. Our rendering techniques optimize bandwidth utilization and provide very high performance.

We have also generalized our computational video approach to include streaming data types other than video. In particular, we have developed a system for reliably multicasting time-critical data to heterogeneous clients using overlay networks. To facilitate intelligent content pruning, we have built systems to dynamically tag real-time data streams with XML descriptions. These descriptions are forwarded to the network where they can be interpreted, filtered, and even relabeled by subsequent application-level XML routers. XML routers perform content-based routing of individual XML packets to subsequent routers or clients based upon filters that describe the information needs of down-stream nodes. We have applied this technology to live radar feeds.

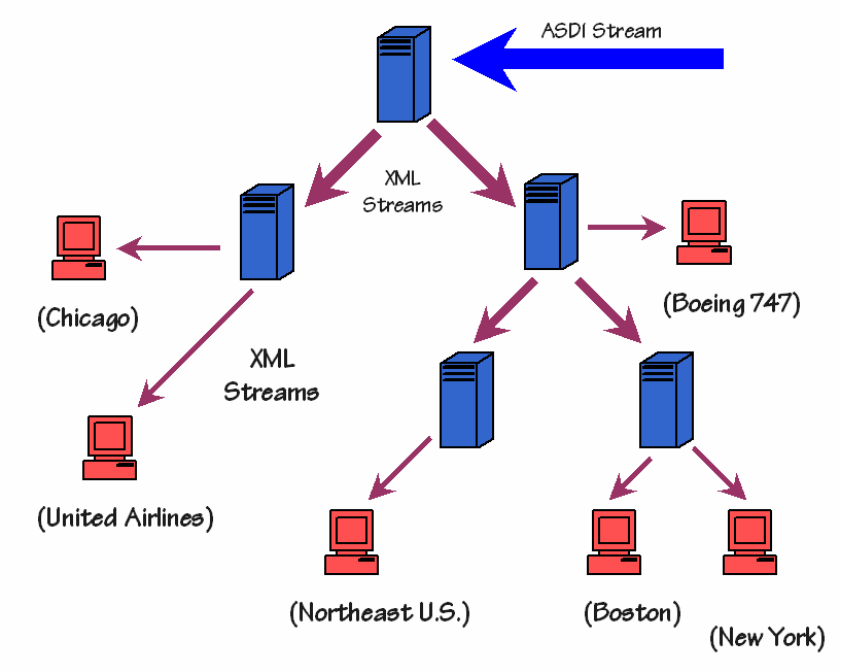


Figure 5. Our content routing system processes real-time data streams, in this case radar data, by adding XML tags. These XML tags are used by application-level routers to process, redirect, filter, and re-label information in the stream for subsequent distribution.

Program Accomplishments:

The following publications, which are provided along with their abstracts, are direct results of this program.

1. Pure Java-based Streaming MPEG Player. Osama Tolba, Hector Briceño, and Leonard McMillan, *SPIE Proceedings Vol. 3528: Multimedia Systems and Applications*, SPIE's Symposium, Boston, November 1998.

We present a pure Java-based streaming MPEG-1 video player. By implementing the player entirely in Java, we guarantee its functionality across platforms within any Java-enabled web browsers, without the need for native libraries. This allows greater use of MPEG video sequences, because the users will no longer need to pre-install any software to display video, beyond Java compatibility. This player features a novel forward-mapping IDCT algorithm that allows it to play locally stored, CIF-sized (352 x 288) video sequences at 11 frames per second, when run on a personal computer with Java "Just-in-Time" compiler. The IDCT algorithm can run with greater speed when the sequence is viewed at reduced size; e.g., performing approximately ¼ the amount of computation when the user resizes the sequence to ½ its original width and height. We are able to play video streams stored anywhere on the Internet with acceptable performance using a proxy server, eliminating the need for large-capacity auxiliary storage. Thus, the player is well suited to small devices, such as digital TV set-top

decoders, requiring little more memory than is required for three video frames. Because of our modular design, it is possible to assemble multiple video streams from remote sources and present them simultaneously to the viewers (i.e. picture-in-a-picture style), subject to network and local performance limitations. The same modular system can further provide viewers with their own customized view of each session; e.g., moving and resizing the video display window dynamically, and selecting their preferred set of video controls.

2. NAIVE - Network Aware Internet Video Encoding, Hector Briceño, Steven Gortler, and Leonard McMillan, *Proceedings of Seventh ACM International Multimedia Conference*, (ACM MULTIMEDIA'99, Orlando, FL, Oct. 30 - Nov. 5, 1999), pp. 251-260.

The distribution of digital video content over computer networks has become commonplace. Unfortunately, most digital video encoding standards do not degrade gracefully in the face of packet losses, which often occur in a bursty fashion. We propose a new video encoding system that scales well with respect to the network's performance and degrades gracefully under packet loss. Our encoder sends packets that consist of a small random subset of pixels distributed throughout a video frame. The receiver places samples in their proper location (through a previously agreed ordering), and applies a reconstruction algorithm on the received samples to produce an image. Each of the packets is independent, and does not depend on the successful transmission of any other packets. Additionally, each packet contains information that is distributed over the entire image. We also apply spatial and temporal optimization to achieve better compression.

3. Image-Based Visual Hulls. Wojciech Matusik, Chris Buehler, Ramesh Raskar, Steven Gortler, and Leonard McMillan, *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000, pp. 369-374.

In this paper, we describe an efficient image-based approach to computing and shading visual hulls from silhouette image data. Our algorithm takes advantage of epipolar geometry and incremental computation to achieve a constant rendering cost per rendered pixel. It does not suffer from the computation complexity, limited resolution, or quantization artifacts of previous volumetric approaches. We demonstrate the use of this algorithm in a real-time virtualized reality application running off a small number of video streams.

4. Dynamically Reparameterized Light Fields, Aaron Isaksen, Leonard McMillan, and Steven Gortler, *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, July 2000, pp. 297-306.

This research further develops the light field and lumigraph image-based rendering methods and extends their utility. We present alternate parameterizations that permit 1) interactive rendering of moderately sampled light fields of scenes with significant, unknown depth variation and 2) low-cost, passive autostereoscopic viewing. Using a dynamic reparameterization, these techniques can be used to interactively render

photographic effects such as variable focus and depth-of-field within a light field. The dynamic parameterization is independent of scene geometry and does not require actual or approximate geometry of the scene. We explore the frequency domain and ray-space aspects of dynamic reparameterization, and present an interactive rendering technique that takes advantage of today's commodity rendering hardware.

5. Polyhedral Visual Hulls for Real-Time Rendering. Wojciech Matusik, Chris Buehler, and Leonard McMillan, *Proceedings of Twelfth Eurographics Workshop on Rendering*, London, England, June 2001, pp. 116-126.

We present new algorithms for creating and rendering visual hulls in real-time. Unlike voxel or sampled approaches, we compute an exact polyhedral representation for the visual hull directly from the silhouettes. This representation has a number of advantages: 1) it is a view-independent representation, 2) it is well-suited to rendering with graphics hardware, and 3) it can be computed very quickly. We render these visual hulls with a view-dependent texturing strategy, which takes into account visibility information that is computed during the creation of the visual hull. We demonstrate these algorithms in a system that asynchronously renders dynamically created visual hulls in real-time. Our system outperforms similar systems of comparable computational power.

6. Unstructured Lumigraph Rendering. Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen, *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, July 2001, pp. 425-432.

We describe an image based rendering approach that generalizes many current image based rendering algorithms, including light field rendering and view-dependent texture mapping. In particular, it allows for lumigraph-style rendering from a set of input cameras in arbitrary configurations (i.e., not restricted to a plane or to any specific manifold). In the case of regular and planar input camera positions, our algorithm reduces to a typical lumigraph approach. When presented with fewer cameras and good approximate geometry, our algorithm behaves like view-dependent texture mapping. The algorithm achieves this flexibility because it is designed to meet a set of specific goals that we describe. We demonstrate this flexibility with a variety of examples.

7. Mesh Based Content Routing using XML. Alex C. Snoeren, Kenneth Conley, and David K. Gifford, ACM Symposium on Operating System Principles, October 2001, pp. 160-173.

We have developed a new approach for reliably multicasting time-critical data to heterogeneous clients over mesh-based overlay networks. To facilitate intelligent content pruning, data streams are comprised of a sequence of XML packets and forwarded by application-level XML routers. XML routers perform content-based routing of individual XML packets to other routers or clients based upon queries that describe the information needs of down-stream nodes.

Our routers use a novel Diversity Control Protocol (DCP) for router-to-router and router-to-client communication. DCP reassembles a received stream of packets from one or more senders, using the first copy of a packet to arrive from any sender. When each node is connected to n parents the resulting network is resilient to n-1 router or independent link failures without repair. Associated mesh algorithms permit the system to recover to n-1 resilience after node and/or link failure. We have deployed a distributed network of XML routers that streams real-time air traffic control data. Experimental results show multiple senders improve reliability and latency when compared to tree-based networks.

8. Efficient View-Dependent Sampling of Visual Hulls. Wojciech Matusik, Chris Buehler, and Leonard McMillan, MIT Technical Memo MIT-LCS-TM-623

In this paper we present an efficient algorithm for sampling visual hulls. Our algorithm computes exact points and normals on the surface of visual hull instead of a more traditional volumetric representation. The main feature that distinguishes our algorithm from previous ones is that it allows for sampling along arbitrary viewing rays with no loss of efficiency. Using this property, we adaptively sample visual hulls to minimize the number of samples needed to attain a given fidelity. In our experiments, the number of samples can typically be reduced by an order of magnitude, resulting in a corresponding performance increase over previous algorithms.

Project Milestones and Accomplishments:

In addition to the publications and technical reports mentioned above, the various results of this research program have been presented to the DARPA community at the following venues:

November 1999, DARPA ITO PI Meeting, HI – We presented live demos of our remote presentation, and Java-based MPEG decoder systems, as part of the work-in-progress demonstration system. Participants at the demo site were able to dynamically change the view of a remote presentation system originating from our graphics lab in Cambridge MA. Participants were able to freely look around the lab.

September 2000, DARPA ITO PI Meeting, San Diego, CA – We presented demonstrations of our image-based visual hull and Naïve video compression systems. We used a four-camera system to synthesize real time virtual models of demo attendees. In addition to demonstrating progress on our research efforts this effort demonstrated the robustness and portability of our system.

Conclusions:

The computational video project has generated a wide range of fundamental research results, which have had a large impact on the graphics and vision research communities. As a measurement of this impact one needs only look at the on-line citation database CiteSeer (http://citeseer.nj.nec.com/cs) to see the number of reference to work funded through this program. For instance, the image-based visual-hull work has been referenced 23 subsequent research papers, and the dynamically re-parameterized light-field work has been cited by 8 other publications.

Appendix A:

Pure Java-based Streaming MPEG Player

Osama Tolba, Hector Briceño, and Leonard McMillan*
Laboratory for Computer Science, MIT

ABSTRACT

We present a pure Java-based streaming MPEG-1 video player. By implementing the player entirely in Java, we guarantee its functionality across platforms within any Java-enabled web browsers, without the need for native libraries. This allows greater use of MPEG video sequences, because the users will no longer need to pre-install any software to display video, beyond Java compatibility. This player features a novel forward-mapping IDCT algorithm that allows it to play locally stored, CIF-sized (352 x 288) video sequences at 11 frames per second, when run on a personal computer with Java "Just-in-Time" compiler. The IDCT algorithm can run with greater speed when the sequence is viewed at reduced size; e.g., performing approximately ¼ the amount of computation when the user resizes the sequence to ½ its original width and height. We are able to play video streams stored anywhere on the Internet with acceptable performance using a proxy server, eliminating the need for large-capacity auxiliary storage. Thus, the player is well suited to small devices, such as digital TV set-top decoders, requiring little more memory than is required for three video frames. Because of our modular design, it is possible to assemble multiple video streams from remote sources and present them simultaneously to the viewers (i.e. picture-in-a-picture style), subject to network and local performance limitations. The same modular system can further provide viewers with their own customized view of each session; e.g., moving and resizing the video display window dynamically, and selecting their preferred set of video controls.

Keywords: Java, MPEG, forward-mapping IDCT, streaming video.

1. INTRODUCTION

Due to the recent explosion of digital video and its applications in the Internet and broadcast media, there is a growing need for Internet-based video players. Some Internet streaming video players already exist, such as RealPlayer by RealNetworks, Inc., but they are platform dependent and require downloading and updating the player code regularly. We believe that players implemented using the new Java Media Framework (JMF) have some limitations too. While JMF implementations can play MPEG-1 video they still rely on native libraries that must be downloaded and installed on the browser's machine.[7] JMF also imposes other limitations; for example, low-level access to picture data is prohibited. We set out to explore whether a player could be implemented entirely in Java. Here are some of the advantages of Java that led us to choosing it for our player, and shortcomings of Java that we have encountered.

Advantages of a Java-based player:

1. Programs written entirely in Java run across platforms within any Java-enabled Web browser or other Java Virtual Machines (JVM' s), without the need for native libraries. This allows greater use of MPEG video sequences, because the users will no longer need to pre-install any software plug-ins to display video. All that is required is Java compatibility, preferably with Just-in-Time (JIT) compiling.
2. Java's small footprint and availability for small devices.
3. Java programs are compact. The size of the Jar file for our minimal MPEG player is 40 kilobytes.
4. Extensive networking capabilities are built into the language, making it easy to write programs that use Internet communication.

Shortcomings of Java:

* Address: 545 Technology Square, Cambridge, MA 02139. E-mail: {tolba, hbriceno, mcmillan}@graphics.lcs.mit.edu

1. Programs written in Java—an interpreted language—run slower than ones written in compiled languages, such as C, even with the aid of JIT compilers which translate Java's byte code into native code.
2. Another performance shortcoming of Java lies within its windowing toolkit, the AWT. In our results we show that the video's update rate is greatly influenced by the rate at which the screen component can load and draw the new image.
3. While Java's and Internet browsers' security restrictions are useful for many applications, they create extra work for programmers wishing to establish read-only network connections. Such connections are commonplace in a Web browser but a Java applet is not allowed to perform them unless the browser's security settings are lowered. We have found the task of lowering the security restrictions cumbersome and decided to augment our applications with a server-side proxy server residing on the same host as the applet that makes the necessary network connection and forwards the data to the applet.
4. Java's application programming interface (API) is rapidly evolving. While providing great enhancements, such pace burdens the programmer with compatibility issues. For example, our initial implementation employed Java's older version 1.0 but was extremely slow to update an animated image. Therefore, the final implementation, which uses version 1.1, is not guaranteed to run on every Java machine.
5. Because Java is new and evolving, different Java machines have widely varied performances. This is most evident in the ten-fold increase in performance when using the JIT compilers.

Many video sources in the Internet can be extremely long and in some cases arbitrarily long as in the case of live video. In the past, players downloaded the entire video sequence before playing it. This initial waiting period limited the broad utilization of video. Recently video players have adopted the strategy of decoding video sequences as they are received; a technique referred to as *streaming*. Streaming has the side-benefit of constant storage requirements at the receiver. Our video player is capable of playing streaming video from either local files or the Internet. Currently we are using the HTTP protocol[3] that is compatible with all web-servers to fetch video from the Internet, but other protocols can be incorporated into our modular framework.

In section 2 we give a description of the system architecture we adopted for the player. Specific details about the inverse discrete cosine transform (IDCT) algorithm employed in this player are given in section 3, while Sections 4 and 5 include the results and discussion. This player was implemented as part of the ongoing *Computational Video* project at MIT.[8]

2. SYSTEM ARCHITECTURE

We have adopted a modular design of the video player based on source-player (producer-consumer) architecture. In this architecture, a single Java canvas or panel is overloaded to act as the player associated with a source object that reads bits from the video stream and decodes the individual frames. Any number of these players can be embedded in a Java applet or application. The applet may also host a panel of video controls resembling a video cassette player that communicates user interaction to all the players in the applet. The player panel has a similar set of controls in the form of a popup menu with additional controls that allow the viewer to change the size and appearance of the video component. Such modular design allows the users to assemble their own customized sessions including any number and shape of video windows and controls. We envision a scenario where a variety of video controls with different styles and functionality is available, perhaps via the Internet. The session can be configured dynamically as well, for example to move or reduce the footprint of a player to expose the video players behind it.

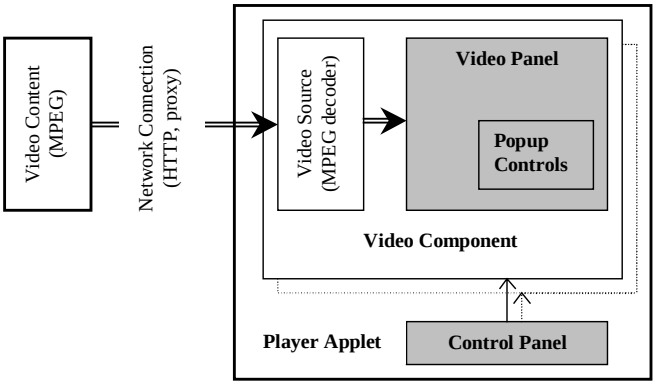


Figure 1. Diagram showing the source-player architecture of the video player, with visual parts shaded.

Under this architecture the video producer can implement any video CODEC, in this case is an MPEG decoder. The decoder reads its bits from an input stream residing on the applet's HTTP server, or any other server via a proxy server as discussed later. Instead of downloading the entire file beforehand, the producer begins decoding the video stream immediately, thereby eliminating the need for large amounts of memory. However, because of the structure of the MPEG stream, which includes forward and backward prediction video frames, the decoder must store two reference frames in addition to the one currently being decoded. Hence, the overall memory requirements of this decoder consist of three YUV frames, double-buffered display, room for lookup tables, and limited input buffer.

The need for a proxy server arises from Java's and Web browsers' security limitations. Most browsers, in their default settings, do not permit Java applets to make network connections, such as the ones needed to play video sequences from other Internet locations. A proxy is a server that acts as an intermediary, forwarding bits from a source server to a destination client that cannot make the direct connection itself. Naturally, the proxy server has a limited buffer in order to reduce network communication and enhance performance. We implemented this server using the C language for efficiency and because it does not affect the player's portability.

3. FORWARD-MAPPING IDCT

MPEG decoding involves many steps, as shown in Figure 2. A straightforward implementation of these procedures in Java would prove to be too slow for practical use. In order to optimize this process it is necessary to identify its bottlenecks. We have found the IDCT computation to be a major one.[†] In this section we describe a novel IDCT algorithm we have developed based on McMillan and Westover's forward-mapping IDCT (FMIDCT) and the table lookup technique that combines the inverse quantizer and the IDCT. The performance gains allow us to decode MPEG streams in Java software without native libraries or hardware acceleration.

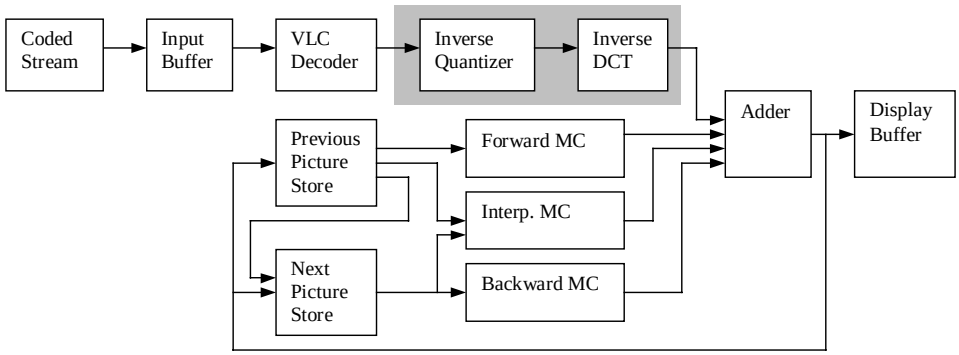


Figure 2. MPEG decoder block-diagram, with shaded area showing where the FMIDCT optimizes the process.

3.1 Background

McMillan and Westover presented an algorithm for computing the IDCT that exploits the sparseness of the typical IDCT input sequence.[4] The procedure involves computing the individual contributions of each input vector element and accumulating its contribution into the final result. This style of computation is commonly referred to as *forward-mapping* evaluation. This technique for the IDCT can be easily derived from the definition of the type II two-dimensional NxN IDCT given below:

$$O_{x,y} = \frac{2}{N} \sum_{v=0}^{N-1} \sum_{u=0}^{N-1} f_u f_v \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2N}\right] I_{u,v}, \quad (1)$$

[†] If this were not true, there would be no significant speedup in decoding predicted (P or B) frames, as opposed to intra-coded (I) frames. In our experiments, I-frames take twice as long to decode as other predicted frames on the average.

$$\text{where } f_i = \begin{cases} \frac{\sqrt{2}}{2} & i = 0 \\ 1 & otherwise \end{cases}$$

This equation can be expressed as a matrix product as follows:

$$\mathbf{O} = \mathbf{C}\,\mathbf{I}\,, \tag{2}$$

where **O** and **I** are column vectors resulting from the row enumeration of $O_{x,y}$ and $I_{u,v}$, and **C** is a N^2xN^2 matrix.

The contribution of each element $i_{u,v}$ of the input vector **I** is the product of $i_{u,v}$ and a particular column within matrix **C**. This column is the unit-valued basis vector $\mathbf{K}_{u,v}$, unique to input $i_{u,v}$. One can easily observe that zero-valued elements of the input vector make no contribution to the output sequence. Furthermore, while each unit-valued basis vector $\mathbf{K}_{u,v}$ has N^2 coefficients there are at most N(N+2)/8 unique coefficients if the sign is ignored. For an 8x8 transform the number of unique coefficients for each of the basis vectors is summarized in Table 1.

v \ u	0	1	2	3	4	5	6	7
0	**1**	**4**	**2**	**4**	**1**	**4**	**2**	**4**
1	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**
2	**2**	**8**	**3**	**8**	**2**	**8**	**3**	**8**
3	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**
4	**1**	**4**	**2**	**4**	**1**	**4**	**2**	**4**
5	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**
6	**2**	**8**	**3**	**8**	**2**	**8**	**3**	**8**
7	**4**	**10**	**8**	**10**	**4**	**10**	**8**	**10**

Table 1. Number of unique coefficients for the basis vectors in an 8x8 transform.

The forward-mapping approach takes advantage of this repetition by performing the minimum number of multiplies, corresponding to the number of unique coefficients within the basis functions, and then performing the appropriate add or subtract into the output vector.

In practice the input sequence presented to the IDCT is generated by a dequantization step. This dequantization requires a unique multiplication for each element in the input sequence, and potentially other non-linear operations. In the FMIDCT the dequantization and the basis function scaling for each element of the input sequence are combined into a single table lookup. For an 8x8 transform this requires an M entry table where each table entry is composed of K elements, where Table 1 determines K. The contents of a table entry reflect both the dequantization and subsequent scaling of the unique coefficients of basis function. McMillan and Westover also described a technique for reducing the number of table elements and deferring their generation. As a result, the direct computation of the two-dimensional IDCT can typically be accomplished with fewer than 10 adds or subtracts and virtually no multiplies per output element, which compares favorably to any other fast IDCT implementation.

3.2 Improvements to the FMIDCT

In the remainder of this section we describe new improvements to the FMIDCT, achieved by exploiting the symmetries of the basis functions. These improvements reduce the required number of additions by a factor of four. It uses a smaller accumulator array that can potentially fit in registers, which greatly enhances the performance. Also, we reduced the size of the lookup tables, improving their hit-rate, and share them between dequantizers.

Each of the two-dimensional basis functions associated with a given input can be classified into one of four symmetries as shown below:

$$\text{Type A: } \begin{bmatrix} Q & H \\ V & D \end{bmatrix}, \text{ Type B: } \begin{bmatrix} Q & -H \\ V & -D \end{bmatrix}, \text{ Type C: } \begin{bmatrix} Q & H \\ -V & -D \end{bmatrix}, \text{ and Type D: } \begin{bmatrix} Q & -H \\ -V & D \end{bmatrix}, \tag{3}$$

where Q is the upper-left quadrant of the basis function, $H = Q\,R$, $V = R\,Q$, $D = R\,Q\,R$, and $R = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$.

In other words, *H* is the horizontal mirror image of *Q*, *V* is its vertical mirror image, and *D* is mirrored in both dimensions. The recognition of these basis function symmetries allows the number of accumulators to be reduced by a factor of four. Table 2 classifies the symmetries of the 64 basis functions in an 8x8 transform.

V \ U	0	1	2	3	4	5	6	7
0	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
1	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**
2	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
3	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**
4	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
5	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**
6	**A**	**B**	**A**	**B**	**A**	**B**	**A**	**B**
7	**C**	**D**	**C**	**D**	**C**	**D**	**C**	**D**

Table 2. Symmetry types for the basis functions in an 8x8 transform.

The improved FMIDCT algorithm requires four $N^2/4$ accumulator arrays, Q_A, Q_B, Q_C, and Q_D, one for each symmetry type. Only the upper left quadrant of the basis vector is computed for any symmetry, thereby reducing the required computation by a factor of four. As each element of the input array is processed its contribution is accumulated into the appropriate array according to Table 2. Finally, the four accumulator arrays are merged to form the intensity values as follows:

$$\begin{aligned} Q_F &= Q_A + Q_B + Q_C + Q_D \\ H_F &= (Q_A - Q_B + Q_C - Q_D)\,R \\ V_F &= R\,(Q_A + Q_B - Q_C - Q_D) \\ D_F &= R\,(Q_A - Q_B - Q_C + Q_D)\,R \end{aligned} \tag{4}$$

Where Q_F, H_F, V_F, and D_F are the four quadrants in a final block of luminance or chroma values. Computing the quadrants in two butterfly stages as shown below can reduce the three operations per output element implied in equation 4:

$$\begin{aligned} B_{1a} &= Q_A + Q_B \quad B_{1b} = Q_A - Q_B \\ B_{2a} &= Q_C + Q_D \quad B_{2b} = Q_C - Q_D \\ Q_F &= B_{1a} + B_{2a} \\ H_F &= (B_{1b} + B_{2b})\,R \\ V_F &= R\,(B_{1a} - B_{2b}) \\ D_F &= R\,(B_{1b} - B_{2a})\,R \end{aligned} \tag{5}$$

Thus the total number of operations required for the improved FMIDCT is $(L/4 + 2)\,N^2$, where L is the number of non-zero coefficients, versus LN^2 for the original implementation. For all values of L greater than 2 the new formulation requires fewer operations. When L equals 2 no more that 2 of the quadrant accumulator arrays will contain nonzero values, thus equation 6 can be simplified such that no more than one operation is required to unfold the symmetries. When L equals 1 the merger requires no accumulation.

3.3 Lookup Tables and Caching Techniques

In the original FMIDCT implementation the dequantization process was assumed to have only two independent variables, the quantized symbol and the quantizer scale. Since the quantizer scale typically varies from element to element of the input vector, the original FMIDCT required N^2 separate tables. The dequantization process, however, is often more complex—containing as many as four independent variables. For instance, in MPEG-1 the dequantizer is defined by four parameters, a 31-level quantizer scale defined for each macro-block, a coefficient quantizer value for each element of the input vector, a binary variable based on whether the block is intra-frame or inter-frame encoded, and the quantized symbol. Implementing the table lookup function as described in the original FMIDCT would result in a factor of 62 increase in table size. In addition to the dramatic increase in storage requirements the hit-rate of the hashing-based lazy-evaluation technique is considerably reduced, thereby increasing the number of multiplies.

Another important property of the underlying basis vectors is there are only S unique sets. This reflects the fact that each of the basis functions with U unique coefficients shares the same U coefficients. So in the case of an 8x8 transform, all the basis functions with ten unique coefficients share in common the same coefficient values as all other basis vectors with ten coefficients. Therefore, it is conceivable that several different combinations of dequantization parameters might resolve into the same value. The improved version of the FMIDCT uses a two-level table lookup for dequantization and the scaling of basis functions. This permits the tables' contents to be shared between dequantizers.

The technique is described as follows. A hashing function, $H(\mathbf{p})$, is applied to the parameter vector **p** containing the four dequantizer parameters. The result is used to index the first level table, called the *dequantizer cache*. Each entry of the dequantizer cache is composed of a key parameter vector **k** and an index i_d into the second table, which stores the scaled basis vectors. If **k** matches **p** then it is used to index one of S tables, one for each of the unique basis vector sets and the scaled basis vector is constructed from the values stored there. If **k** does not match **p** then the appropriate dequantizer is invoked with parameter vector **p**, the dequantized value replaces i_d and **k** replaced by **p**. The dequantized value i_d is then used as an index into the second level table as before. The key values in the dequantization cache should be first initialized to an invalid state (we have used the fact that the quantizer symbols cannot have a value of zero). The process is shown with sample code in the Appendix.

4. IMPLEMENTATION & RESULTS

We have implemented an MPEG-1 video player in as a Java applet that utilizes our improved FMIDCT algorithm. This applet is capable of decoding and displaying 176 x 144 video sequences at rates of approximately 28 frames per second, and 352 x 240 sequences at 11 frames per second using Internet Explorer 4.0 on a Pentium computer with PII 266 MHz processor. Table 3 gives the results for sample sequences gathered from various Internet locations. While the player is capable of playing them directly from their respective sources, the results shown here are for sequences that have been downloaded to local storage before being played. Had they been played directly across the Internet, the frame rates would have been slower reflecting network bottlenecks rather than decoding time. The average performance of decoding and displaying an MPEG video sequence seems to be on the order of 670,000 pixels per second; with a three-fold increase in the performance when the pictures are decoded but not displayed, achieving 1.8 million pixels per second. This illustrates our previous suggestion that Java's display technology hinders the performance of the player, consuming two-thirds of the total time needed to decode and display the pictures. Future releases of Java promise to reduce this problem. In practice, Internet broadcasts take network bandwidth limitations into consideration, limiting the dimensions and frame rates to those manageable by our player. For instance, most Internet broadcast video has an image size of 176 x 144 and frame rates of 15 frames per second or less. It is worth noting that we have encountered other Java applets that play MPEG video via the Internet, to which our player compares favorably.[11,12]

Video Clip	Width	Height	Pixels	Decode rate	Pixels/sec	Display rate	Pixels/sec
30way	160	112	17,920	115	2,060,800	36.88	660,890
benylin	176	144	25,344	99.5	2,521,728	28.71	727,626
cart	240	180	43,200	28.7	1,239,840	16.25	702,000
daimler	368	272	100,096	14	1,401,344	6.05	605,581
glacier	200	100	20,000	90.4	1,808,000	38.2	764,000
hill	240	180	43,200	40.2	1,736,640	16.75	723,600
ligther	176	144	25,344	73.1	1,852,646	25.46	645,258
llupanav	160	128	20,480	79.7	1,632,256	24.86	509,133
mjackson	160	120	19,200	110.3	2,117,760	32.88	631,296
ts	352	240	84,480	15.7	1,326,336	7.7	650,496
wg_wt_1	304	224	68,096	29.8	2,029,261	10.9	742,246
Average					1,793,328		669,284

Table 3. Frame rates achieved with various sample video sequences.

Our modular source-player design enabled us to implement a picture-in-a-picture player. Figure 3 shows a screen capture of an applet playing two video streams simultaneously. The user can resize and move each window dynamically via mouse interaction. This applet and other demonstrations are available at our Internet site.[8]

Figure 3. Single player with textual video controls (left) and a picture-in-a-picture example with visual controls (right).

5. CONCLUSIONS

We have presented an efficient streaming MPEG-1 video player implemented entirely in Java, which eliminates the need to pre-install native software and is well suited to small devices. The player owes its efficiency to an improved forward-mapping IDCT algorithm described here as well. Our modular design provides viewers with their own customized session, which we demonstrated in the picture-in-a-picture example.

Further tests and enhancements to this player include the implementation of the trade-off between quality and speed feature, described by McMillan and Westover. For example, the IDCT algorithm can run with greater speed when the video is viewed at reduced size, performing approximately ¼ the amount of computation when the user resizes the video to ½ its original width and height. It is also possible to use this feature to improve the frame rates of video with large pictures by playing them at reduced size or reduced quality. However, this feature was not envisioned with MPEG video in mind and some difficulties exist in using it here. A crude application of this feature to intra-coded frames would result in compounded degradation of predicted frames, introducing new artifacts into the decoded stream. Another feature that we would like to investigate is rate control. Currently, we are playing the sequences at the fastest achievable rates, which are generally below the sequences' original frame rates. It is possible to drop frames by avoiding decoding them, but the structure of MPEG video makes this

feature problematic because of inter-frame dependencies. For example, if we decided to decode all intra-coded frames in anticipation that other frames will depend on them, we may encounter sequences made up entirely of I-frames, for which this technique will fail to speed the play back. Alternatively, we could defer decoding P or I-frames that we decide to skip until we encounter other frames that depend on them. Such method is complicated, demands more memory, and does not guarantee speed either. Finally, we are currently using this framework to design and implement other players in the context of the Computational Video project, such as a players for motion-JPEG video and other special purpose video.

6. ACKNOWLEDGEMENTS

This work is supported by DARPA agreement number F30602-97-1-0283. The video sequences used in our demonstrations and statistics were downloaded from various Internet sources reached via *MPEG.ORG*'s Web site.[10]

7. REFERENCES

1. Kamanagar, F. A. and K. R. Rao, "Fast Algorithms for the 2-D Discrete Cosine Transform", *IEEE Trans. On Computers*, vol. C-31, no. 9, pp. 899-906, Sep 1982.
2. LeGall, D. "MPEG: A Video Compression Standard for Multimedia Applications", *Com. of the ACM*, vol. 34, no. 4, pp. 46-58, April 1991.
3. Fielding R. et al, "*Hypertext Transfer Protocol - HTTP/1.1*", Internet RFC 2068, January 1997.
4. McMillan, Leonard and Lee Westover, "A Forward-Mapping Realization of the Inverse Discrete Cosine Transform", *Proceedings of the Data Compression Conference (DCC '92)*, IEEE Computer Society Press, March 24-27, pp. 219-228, 1992.
5. Rao, K.R. and P. Yip. *Discrete Cosine Transform: algorithms, advantages, applications*. Academic Press, Boston, 1990.
6. Vetterli, M. and H. Nussbaumer, "Simple FFT and DCT Algorithms with Reduced Number of Operations", *Signal Processing*, vol. 6, pp. 267-278, Aug 1984.
7. http://java.sun.com
8. http://compvid.lcs.mit.edu/cv/
9. http://bmrc.berkeley.edu/projects/mpeg/
10. http://www.mpeg.org
11. http://www.dcc.uchile.cl/~chasan/MPEGPlayer.zip
12. http://rnvs.informatik.tu-chemnitz.de/~ja/MPEG/JITVERS/MPEG_Play.html

APPENDIX

The following Java code fragment demonstrates the first stage of table lookup associated with the dequantizer cache:

```
// Dequantizer cache
private int cacheKey[] = new int[4096];
private int cacheEntry[] = new int[4096];

// Each entry in icoeff packs two values: position in the 8x8 block and quantized DCT coefficient
private void dequantize( int qscale, int []icoeff, byte []qmatrix, int type) {
    // Store output in place. We will use the results in the IDCT method
    // (not shown here) to look up the scaled basis vectors
    int []ocoeff = icoeff;

    int i, j, k, index, key, level, qval;

    // DC of intra-coded block
    j = k = 0;
    if (type == 0) {
        level = icoeff[j++];
        level >>= 8;
        ocoeff[k++] = (level + 2048) * 32 << 8;
    }

    // Process all input coefficients
    while ((level = icoeff[j++]) != 0) {
        i = level & 63;                 // position in 8x8 block
        level >>= 8;                    // quantized DCT coefficient
        qval = qmatrix[i];              // quantization value for this position

        // Pack the four-parameter vector p into a single integer
        key = (level << 16) | (qval << 8) | (type << 5) | qscale;
```

```
        // Determine hashed index H(p)
        index = ( level + ((qval - 16) << 4) + ((qscale - 8) << 8) + (type << 11)) & 0xfff;

        if (cacheKey[index] == key) {   // k stored in cache matches p
            level = cacheEntry[index];  // retreive dequantized coefficient from cache
        } else {
            // Invoke appropriate dequantizer
            if (type == 0) level = (level * qscale * qval) >> 3;
            else level = ((2 * level + ((level >> 31) | 1)) * qscale * qval) >> 4;
            if (level == 0) continue;
            if ((level & 1) == 0) level -= (level >> 31) | 1;

            // Clamp dequantized coefficient to interval [-2048, 2047]
            if (level > 2047) level = 2047;
            else if (level < -2048) level = -2048;

            // Convert dequantized coefficient into index of scaled basis vector
            // in the second table (we add 2048 to avoid negative indices)
            level = (level + 2048) * 32;

            // Store entry in dequantizer cache
            cacheKey[index] = key;      // replace k by p
            cacheEntry[index] = level;  // replace i_d with dequantized coefficient
        }
        ocoeff[k++] = (level << 8) | i; // pack level and index into output coeff
    } // No more input coefficients
    ocoeff[k] = level;
}
```

Appendix B:

NAIVE – Network Aware Internet Video Encoding

Hector M. Briceño
MIT
hbriceno@lcs.mit.edu

Steven Gortler
Harvard University
sjg@cs.harvard.edu

Leonard McMillan
MIT
mcmillan@lcs.mit.edu

Abstract

The distribution of digital video content over computer networks has become commonplace. Unfortunately, most digital video encoding standards do not degrade gracefully in the face of packet losses, which often occur in a bursty fashion. We propose an new video encoding system that scales well with respect to the network's performance and degrades gracefully under packet loss. Our encoder sends packets that consist of a small random subset of pixels distributed throughout a video frame. The receiver places samples in their proper location (through a previously agreed ordering), and applies a reconstruction algorithm on the received samples to produce an image. Each of the packets is independent, and does not depend on the successful transmission of any other packets. Additionally, each packet contains information that is distributed over the entire image. We also apply spatial and temporal optimization to achieve better compression.

1 Introduction

With the advent of the internet, the distribution of digital video content over computer networks has become commonplace. Unfortunately, digital video standards were not designed to be used on computer networks. Instead, they generally assume a fixed bandwidth and reliable transport from the sender to the receiver. However, for the typical user, the internet does not make any such guarantees about bandwidth, latency or errors. This has lead to the adaptation or repackaging of existing video encoding standards to meet these constraints. These attempts have met with varying levels of success. In this paper we propose to design a new video encoding algorithm specifically for computer networks from the ground up.

The internet is a heterogeneous network whose basic unit of transmission is a packet. In order to assure scalability, the internet was designed as a best effort network - i.e. it makes no guarantees that a packet sent by a host will arrive at the receiver or that it will be delivered in the order that it was sent. This also implies that it makes no guarantees on the latency of the delivery.

A video encoding system designed for computer networks would ideally satisfy the following requirements. The transmitted data stream should be tolerant to variations in bandwidth and error rates along various networking routing paths. A given data stream should also be capable of supporting different qualities of service. Where this quality of service might be dictated by local resources (such as CPU performance) or the other user requirements. These requirements are only partially satisfied by existing video encoding systems. In this paper we propose a flexible video encoding system that satisfies the following design goals:

- The system must allow for broadcast. We would like a system where video can be transmitted to a large audience in real time with no feedback to the source. This allows for arbitrary scalability.
- The network can arbitrarily drop packets due to congestion or difference of bandwidths between networks or receivers. Since this system is targeted to error prone networks, it must perform well under packet losses.
- The sender should be able to dynamically vary the bandwidth and CPU requirements of the encoding algorithm. In order to guarantee a quality of service variations in bandwidth may be necessary. For instance, at scene changes or during a complex sequence. Variations in bandwidth could also occur due to resource limitations at the source such as channel capacity and CPU utilization, or by a policy decision.
- The receiver should be able construct a reasonable approximation of the desired stream using a subset of the data transmitted. Furthermore, the receiver may also intentionally ignore part of the data received to free up resources in exchange for reduced quality.
- The quality of the video should degrade gracefully under packet loss by the network or throttling by the sender or the receiver.
- Variations in the algorithm should support a wide range of performance levels, from small personal appliances to high-end workstations.
- Users should be able to quickly join a session in progress.

These goals place severe constraints on how the system can be built.

We consider packets as the basic unit of network transmission [13]. A video frame generally spans many packets. System throughput and quality are affected by throttling packets at the sender,

packet loss in the network, and ignoring of packets at the receiver. Therefore, we choose to regard packets as atomic in our system design. For scalability and error handling we avoid packets that contain prioritized data or interdependencies, such as the clustering of data or differential encoding. These goals motivate our design principles:

Globalness – Individual packets should contain enough information to reconstruct the whole image. They also should be additive - each additional packet increases the reconstructed image quality. Conversely, for each packet that is dropped by the sender, network or receiver, the image quality degrades.

Independence – All packets are independent of each other; any one of them can be dropped without abrupt changes in quality, and in many cases we can process them out of order.

These principles are quite different than current video encoding systems. Typical video encoding algorithms (i.e. H.263 [1] or ISO MPEG), use compression and encoding techniques that make packets interdependent; when one packet is lost, all other packets that are related to it lose their usefulness.

We propose an encoding system that scales well with respect to the sender's performance, the number of receivers, and the network's performance. This system degrades gracefully under packet loss. Briefly stated: the encoder sends packets that consist of a small random subset of pixels distributed throughout a video frame. The receiver places samples in their proper location (through a previously agreed ordering), and applies a reconstruction algorithm on these samples to produce an image. Notice that since each packet contains a small random subset of the image, there is no ordering or priority for packets. We also apply spatial and temporal optimization to achieve better compression without compromising our global and independence principles.

Many other researchers have shown that there is an inherent tradeoff between the amount of compression and the degree of robustness to data loss [14]. Our work is no exception; our achieved image quality at a given level of compression is below the best known channel encoders. For this price, we obtain the ability to reconstruct images even when receiving one packet per frame. Finding fair ways to measure this tradeoff remains as future work.

2 Previous Work

Video encoding algorithms specifically tailored for the internet have been previously proposed. ISO MPEG-1 provides high compression ratios, and it allows for bitstream resynchronization using slices. Generally slices span multiple packets, and few encoders make an effort to align slices within packet boundaries. The variable length encoding and difference encoding used by MPEG-1 is very effective in reducing the bitrate, but both techniques make assumptions about what has been previously received. If these assumptions are wrong (caused by packet loss) [8], artifacts will develop in the new frame. Other discrete cosine transform (DCT) based algorithms like H.261, have been successfully adapted for use in computer networks by using a technique sometimes called "conditional replenishment" [21]. The idea is, that instead of encoding the differences from previous frames, they either keep old blocks or entirely replenish new blocks independently encoded. These techniques require that all blocks are replenished within a specified period of time. During heavy packet losses, important areas may not be updated until the losses subside. This is an all or nothing approach: a block will completely reach its new state or not change at all.

Layering approaches have partly alleviated this last problem. Algorithms like L-DCT [2] and PVH [21], use a base channel to encode a low quality representation of the block; and use additional channels to encode enhancement information to reproduce a more faithful block. Because enhancement layers usually depend on the base layered being received, when the base layer packets are lost, the block cannot be updated at all.

Error handling can also be incorporated into the network layer. By using error correcting codes, or retransmission based schemes, errors can be minimized or eliminated, as to create the illusion of a reliable network stream. Open-loop approaches [32] (i.e. those that don't require feedback) such as, Forward Error Correction (FEC), eliminate errors when they are well characterized. Unfortunately, these systems must include enough redundancy in advance to deal with the worst-case packet loss rate scenario. This leads to inefficiencies. The overhead for error correction also increases total network load. Thus the entire network is taxed due to the worse performing route [26, 12]. The alternative is to use a closed-loop approach. Close-loop approaches [28, 25, 7, 33] , where the receivers request the retransmission of lost packets, have the drawback of higher latency and are difficult to scale [6, 4]. Additionally, since packet loses generally occur during congestion, these requests and subsequent retransmissions can make matters worse.

Robustness to data loss can be achieved using multiple description coding (MDC) [23, 29, 16]. MDC coders build correlation between the symbols allowing for good reconstruction from subsets of the data. Much of the previous work has dealt with two-channel coding [23], which can withstand the loss half of the transmitted data. There has also been some preliminary work on many-channel coding [16, 29]. One can think of the NAIVE encoding as an extreme example of MDC, where no decorrelating transform is applied to the original pixel data, and pictures can be reconstructed from any received data.

The algorithm we propose bears many resemblences to work in error concealment [3, 11, 34, 31]. While most error concealment techniques are built upon existing standards, our technique proposes an entirely novel encoding scheme. Our encoding scheme is tolerant to bursty errors, and does not require resynchronization. Our reconstruction algorithm is fast, and makes no a-priori assumptions about the existance of specific nearby blocks or pixels.

3 The Algorithm

The Network Aware Internet Video Encoding (NAIVE) system sends a random subset of samples for each video frame and reconstructs the frame at the receiver. The random samples are distributed across one or more network packets. Given a sufficiently uniform sampling distribution, each packet can be considered as a subsampled version of the original image. Thus, each packet satisfies our globalness objective. Samples are selected in a random sequence in order to hide errors caused by packet loss and to reduce aliasing artifacts such as blockiness at low sampling densities [22]. If packets of samples are lost, the degradation is distributed evenly throughout the reconstruction instead of being localized as is typical of the sequentially encoded blocks used in other compression methods. Furthermore, the reconstruction artifacts due to packet loss should lead to an apparent loss in resolution (blurring) rather than introduce spurious structure as would be expected from an uniform subsampling. Such structure is generally visible even when using higher order reconstruction filters.

Following our design principles, each packet contains samples

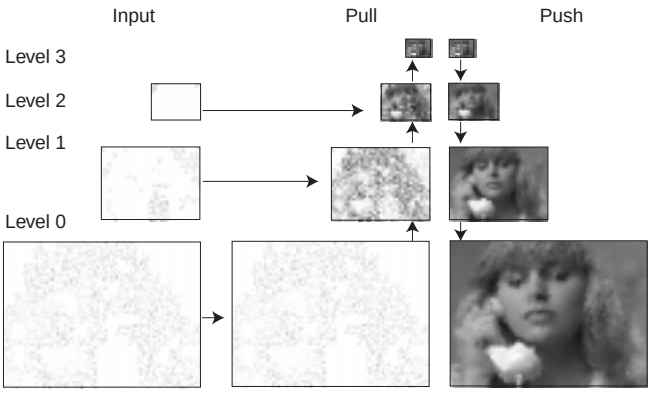


Figure 1: Grayscale *Susie* image pyramid reconstruction. The input samples are located in multiple levels of the pyramid. Notice that input samples in level 1 and 2 correspond to the background and smooth regions of the image.

uniformly distributed throughout the whole image, and independent of any previous packet sent. Our encoding system allows for arbitrary packet loss, thus there is no guarantee that the client has received any particular set of image information. This presents us with the problem of reconstructing an image from irregularly spaced samples.

3.1 Image Reconstruction

A viable solution to this image reconstruction problem must have the following features:

- The method must run at frame rate. Thus, it is too expensive to solve systems of equations (as is done when using global spline methods [30, 19]) or to build spatial data structures (such as a Delauney triangulation [24]).
- The method must deal with spatially scattered samples. Thus we are unable to use standard interpolation methods, or Fourier-based sampling theory.
- The method must create reconstructions of acceptable quality.

In this paper we adapt the pull-push algorithm of Gortler et al. [15]. This algorithm is based on concepts from image pyramids [9], wavelets [20] and subband coding [18], and it extends earlier ideas found in [10] and [22]. The algorithm proceeds in two phases called pull and push. During the first phase, pull, a hierarchical set of lower resolution data sets is created in an image pyramid. Each of these lower resolution images represents a "blurred" version of the input data; at lower resolutions, the gaps in the data become smaller (see pull column in figure 1). During the second phase, push, this low resolution data is used to fill in the gaps at the higher resolutions (compare level 2 pull and push in figure 1). Care is taken not to destroy high resolution information where it is available. Figure 2 shows the reconstruction of the lenna grayscale from 5% and 22% of the original pixels.

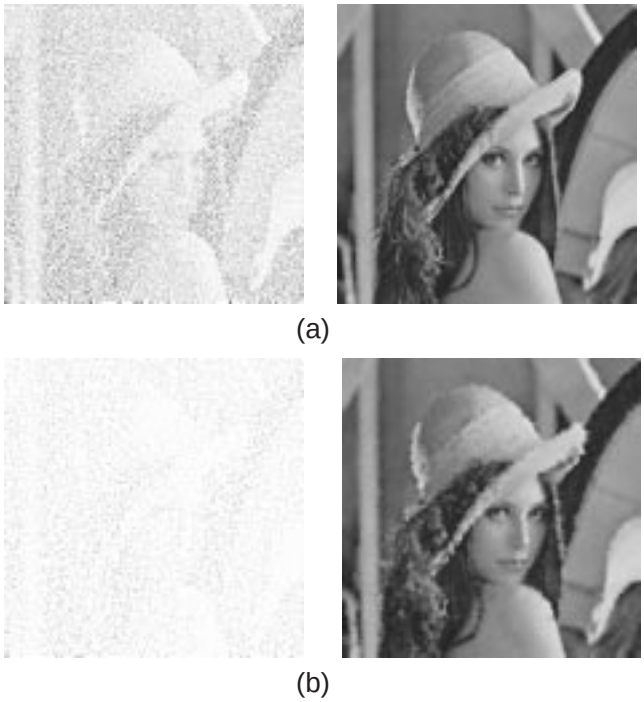


Figure 2: Grayscale lenna image samples and reconstruction. Using 22% original pixels (a), and using 5% of original pixels (b). The images in the left column show the input pixels. The right column shows our reconstruction

3.1.1 Organization

The algorithm uses a hierarchical set of image pixels with the highest resolution labeled 0, and lower resolutions having higher indices. Each resolution has 1/2 the resolution in both the horizontal and vertical dimensions. For our 320 by 240 images, we use a 5 level pyramid. Associated with the ij'th pixel value $p^r_{i,j}$ at resolution r is a weight $w^r_{i,j}$. These weights, representing pixel confidence, determine how the pixels at different resolution levels are eventually combined.

3.1.2 Initialize

During initialization, each of the received pixels is used to set the associated pixel value $p^0_{i,j}$ in the high resolution image, and the associated weight $w^0_{i,j}$ for this pixel is set to f. f is the value chosen to represent full confidence. The meaning of f is discussed below. All other weights at the high resolution are set to 0.

3.1.3 Pull

The pull phase is applied hierarchically, starting from the highest resolution and going until the lowest resolution in the image pyramid. In this pull phase, successive lower resolution approximations of the image are derived from the adjacent higher resolution by performing a convolution with a discrete low pass filter $\tilde{h}$. In our sys-

tem, we use the "tent" sequence. $\tilde{h}[-1..1] \times [-1..1]$:

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$$

The lower resolution pixels are computed by combining the higher resolution pixels using $\tilde{h}$. One way to do this would be to compute

$$\begin{array}{lll} w_{i,j}^{r+1} & := & \sum_{k,l} \tilde{h}_{k-2i,l-2j}\, w_{k,l}^{r} \\ p_{i,j}^{r+1} & := & \frac{1}{w_{i,j}^{r+1}} \sum_{k,l} \tilde{h}_{k-2i,l-2j}\, w_{k,l}^{r}\, p_{k,l}^{r} \end{array} \quad (1)$$

This is equivalent to convolving with $\tilde{h}$ and then downsampling by a factor of two.

This computation can be interpreted as follows: Suppose we have a set of continuous tent filter functions associated with each pixel in the image pyramid. Suppose $\tilde{B}_{i,j}^{0}(u,v)$ is a continuous piecewise bilinear linear tent function centered at i, j and two units (high resolution pixels) wide, $\tilde{B}_{i,j}^{1}(u,v)$ at the next lower resolution is a tent function centered at $2i, 2j$ and is four units (high resolution pixels) wide, $\tilde{B}_{i,j}^{2}(u,v)$ at the next lower resolution is a tent function centered at $4i, 4j$ and is 8 units wide, and so on. These continuous functions are related using the discrete sequence $\tilde{h}$:

$$\tilde{B}_{i,j}^{r+1}(u,v) = \sum_{k,l} \tilde{h}_{k-2i,l-2j} \tilde{B}_{k,l}^{r}(u,v)$$

This means that one can linearly combine finer tents to obtain a lower resolution tent. The desired multiresolution pixel values can be expressed as an integral over an original continuous image $P(u,v)$ using the $\tilde{B}_{i,j}^{r}(u,v)$ as weighting functions:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} du\, dv\;\; \tilde{B}_{i,j}^{r}(u,v) P(u,v) \quad (2)$$

If one approximates this integral with a discrete sum over the received pixel values, one obtains

$$w_{i,j}^{r} p_{i,j}^{r} = \sum_{k,l} \tilde{B}_{i,j}^{r}(k,l) p_{k,l}^{0} w_{k,l}^{0} \quad (3)$$

where

$$w_{i,j}^{r} = \sum_{k,l} \tilde{B}_{i,j}^{r}(k,l)\, w_{k,l}^{0}$$

It is easy to show that the values computed by Equation 3 can be exactly and efficiently obtained by applying Equation 1 hierarchically.

This method creates good low resolution images when the original samples are uniformly distributed. But when the original samples are unevenly distributed, Equation 3 becomes a biased estimator of the desired low resolution value defined by Equation 2 for it overly emphasizes the over sampled regions. Our solution to this problem is to replace Equation 1 with:

$$\begin{array}{lll} w_{i,j}^{r+1} & := & \sum_{k,l} \tilde{h}_{k-2i,l-2j}\, \min(w_{k,l}^{r}, f) \\ p_{i,j}^{r+1} & := & \frac{1}{w_{i,j}^{r+1}} \sum_{k,l} \tilde{h}_{k-2i,l-2j}\, \min(w_{k,l}^{r}, f)\, p_{k,l}^{r} \end{array} \quad (4)$$

The value f represents full confidence, and the min operator is used to place an upper bound on the degree that one image pyramid pixel corresponding to a highly sampled region, can influence the total sum. Any value of $1/16 \leq f \leq 1$ creates a well defined algorithm. If f is set to one, then no saturation is applied, and this equation is equivalent to Equation 1. If f is set to $1/16$, then even a single sample under the sum is enough to saturate the computation for the next lower resolution. In the system we have experimented with many values, and have obtained the best results with $f = 1/8$. Although complete theoretical analysis of the estimator in Equation 4 has yet to be completed, our experiments show it to be far superior to Equation 1. Figure 3 shows the reconstruction of the lenna grayscale image with 10% of its samples reconstructed using (a) f = 1, (b) f = 1/8.

(a) (b)

Figure 3: Grayscale lenna test image reconstruction with 10% of samples: (a) using f = 1, (b) f = 1/8

The pull stage runs in time linear in the number of pixels summed over all of the resolutions. Because each lower resolution has half the density of pixels, the computation time can be expressed as a geometric series and thus this stage runs in time linear in the number of high resolution pixels at resolution 0.

3.1.4 Push

The push phase is also applied hierarchically, starting from the lowest resolution in the image pyramid, and working to the highest resolution. During the push stage, low resolution approximations are used to fill in the regions that have low confidence in the higher resolution images. If a higher resolution pixel has a high associated confidence (i.e., has weight greater than or equal to f), we disregard the lower resolution information for that high resolution pixel. If the higher resolution pixel does not have sufficient weight, we blend in the information from the lower resolution.

To blend this information, the low resolution approximation of the function must be expressed in the higher resolution. This is done using an interpolation sequence also based on the tent sequence but with a different normalization: $h[-1..1] \times [-1..1]$:

$$\begin{bmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 1 & 1/2 \\ 1/4 & 1/2 & 1/4 \end{bmatrix}$$

Push is done in two steps: we first compute temporary values

$$tp_{i,j}^{r} := \sum_{k,l} h_{i-2k,j-2l}\, p_{k,l}^{r+1}$$

This computation is equivalent to upsampling by a factor of 2 (adding 0 values), and then convolving with h. These temporary values are

now ready to be blended with the p^r values already at level r, using the w^r as the blending factors.

$$p^r_{i,j} \quad := \quad (1 - \frac{w^r_{i,j}}{f})\, tp^r_{i,j} + \frac{w^r_{i,j}}{f}\, p^r_{i,j}$$

analogous to the "over" blending performed in image compositing [27].

3.1.5 Compression in the NAIVE Framework

To some extent, NAIVE achieves both compression and resiliency by relying on a random subset of samples from an image to reconstruct the missing information. However, neither the selection nor reception of the samples is related to the specific content of the transmitted image. Since the goal of any compression algorithm is the elimination of redundancy in the target signal, we have also developed techniques to exploit the specific contents of a given video stream to achieve greater compression.

In particular, video sequences commonly exhibit significant spatial and temporal correlations that are generally concentrated in lower frequency ranges. At first glance it would appear that a random sampling strategy, like the one used in NAIVE, runs counter to any effort to reduce spatial and temporal correlation (since randomizing a correlated function tends to decorrelate it). However, if the notion of a sample is expanded to include not only pixels from the highest resolution level of the pyramid hierarchy, but also the subsequent lower resolution levels, significant reductions in spatial correlation can still be achieved. Likewise, if the persistence of a given sample from the reconstruction pyramid is lengthened from a single frame period to multiple frame intervals, similar temporal reductions are also possible.

Often there are cases when an image encoder benefits from transmitting only low-resolution information about some region. Perhaps that region contains little or no high frequency detail, or perhaps the region is considered insignificant and the current instantaneous bandwidth available does not support the transmission of a full resolution image. To accommodate this ability our algorithm allows the encoder to insert lower resolution samples directly into an appropriate level of the pull-push image pyramid, $p^r_{i,j}$ for $r > 0$. When low-resolution samples are received they are placed directly into the reconstruction pyramid at the appropriate resolution. Also, the "pulling" of higher resolution samples onto a lower-resolution sample is suppressed. In order to effectively apply this capability both perceptual and information theoretic concerns should be considered. Thus, as is typical of most digital video compression methods, there is a considerable art to making the best use of this capability. More details about how multi-resolution samples are encoded are given in subsection 4.1.

In video sequences image regions can change slowly. Our system takes advantage of this temporal coherence by allowing pixels from previous frames to be included in the pull-push reconstruction process. The persistence of a given sample is controlled by two mechanisms. First, all samples are aged at a constant rate with newer samples superceding older ones. After a sample's age limit is reached, it no longer takes part in the image reconstruction process. Secondly, entire regions, or blocks, of old samples can be invalidated. This invalidation is typically used in areas of rapid motion or at scene changes. There are many tradeoffs to be considered when using these methods. More information about the aging and invalidation of samples is described in subsection 4.2.

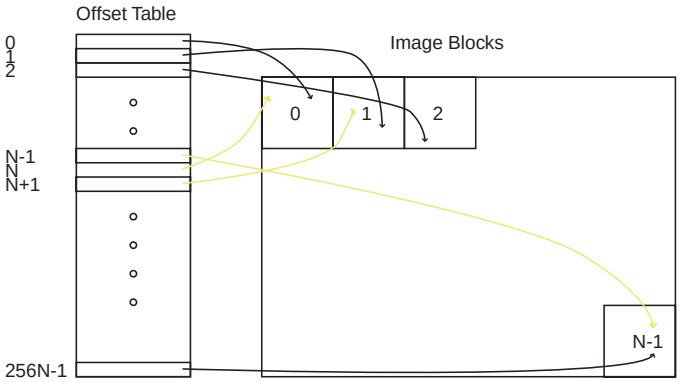


Figure 4: Offset Table: There are N 16x16 blocks in the image. The i'th entry points to a sample in block number i modulo N. On any selection of N consecutive entries, there is a sample from every block

3.2 Packetization

The pull-push algorithm provides a means of reconstructing an image from non-uniform samples. From our principle of globalness we need to pick samples from the whole image. And these have to be selected at random to avoid visible artifacts and to allow the appearance of simultaneous update everywhere in the image [5]. We guarantee coverage of the whole image by dividing it into 16x16 blocks and making succesive passes over the image selecting one random sample from each block on each pass.

In order to minimize the information transmitted, the sender and the receiver agree on the ordering of samples, such that the sender only needs to send the location of the first sample in a packet. This is done as follows. The image is split into 16x16 blocks, this means that there are 256 samples per block. Say there are N blocks in an image. We generate a table, called the "offset table", that has 256*N entries. The i'th entry in the table points to a sample in block number *i mod N*. The first entry contains the coordinate of a random sample in the first block; the second entry contains the coordinate of a sample in the second block; The N+1th entry contains the location of a sample again in the first block. The random ordering of the samples within a block is established by assigning a pseudo-random number to each pixel. The pixels are then sorted into a list according to this random number. The offset table can then be constructed by selecting a pixel from each of the N lists. The sender and receiver are synchronized through the transmission of a seed for the random number generator. With the seed and frame size information the receive can construct the offset table. This is the only information that must be transmitted via a reliable protocol such as TCP/IP.

This ordering guarantees that if we pick N consecutive samples, they will span the whole image without large clusters. Additionally, we can compute the block that a sample belongs from its table offset modulo N. See figure 4.

The reconstruction explained so far applies to a grayscale image. This same idea can be extended to the chrominance components of color images. We encode color images by sampling the chrominance components at a resolution 1/4 of the luminance image, similar to MPEG. To encode them, we maintain another offset table with 8x8 blocks to correspond to the 16x16 blocks of the luminance components. We encode the chrominance samples inde-

Frame Number	#UV samples	Offset UV samples	Offset Y samples	UV samples	Y samples

Figure 5: Packet Format

pendently of the luminance samples.

We need to send very little overhead information with each packet. Each packet consists of: the frame number; table offset of first chrominance sample, number of chrominance samples, and the samples themselves; and table offset of first luminance sample, with the remaining of the packet filled with luminance samples (see figure 5). We use 1024 bytes as our default packet size. This structure satisfies our global and independence properties. If a packet has more than N luminance samples (where N is the number of blocks in a frame), then there will be one sample in every block of the image guaranteed by the way we traverse the offset table.

4 Enhancements

The baseline approach described above works well for images whose details are uniformly distributed throughout the whole image. Most images, though, have localized regions of detail. And most sequences bear a high level of temporal coherency across frames. We can take advantages of these characteristics to produce better quality video with the same or less amount of data.

4.1 Spatial Locality

In image regions with mostly low frequency content, our encoding system allows us to directly transmit lower resolution samples, and the receiver can insert these directly into lower resolution pyramid levels.

In our encoding system, we encode the sample value and resolution level in the same byte. We use 7 bits of precision for level 0 samples, and 6 bits of precision for level 1 and level 2 samples. If the least significant bit is 0, the sample is a level 0 sample; if the least significant bits is 01 or 11 the sample is a level 1 or level 2 sample respectively. With this change we keep the packet structure unchanged, except for how sample values are interpreted.

Samples that are inserted at lower resolution levels, correspond spatially to many more samples at finer levels. Thus, when a low resolution sample is sent, fewer higher resolution samples are needed for that block.

To manage the bookkeeping for this information, we use a special table, called the SKIP TABLE. There is a SKIP TABLE entry for each block. The SKIP TABLE contains the encoder/decoder agreed upon number of samples for this block that will be skipped. When a packet is received, all entries in the SKIP TABLE are initialized to 0; thus each block is guaranteed to have one sample. When a sample is inserted into a lower resolution level, we load the skip table entry for that block, with a predefined constant, agreed upon by the sender and the receiver. In our system, when a sample is sent for level 1, we skip the next 3 samples for this block. When a sample is sent for level 2, we skip the next 15 samples for this block.

Each time that block occurs in the sequence we inspect the skip table entry to see if it is non-zero, if it is, we decrement the skip table, and go to the next block without reading a sample from the packet. Otherwise, we insert the current sample into the block according to the offset table entry.

4.2 Temporal Locality

Temporal locality can be exploited even when packets are independent of each other. MPEG and H.261 exploit temporal locality by reusing block of pixels that are closely located in the previous frame, encoding this location and their difference. In our approach, we don't make any assumptions about the previous frame or what packets the receiver has processed. We simply take advantage of the fact that pixels in a block may not change significantly across many frames, in which case, we reuse them to reconstruct a higher quality image. In NAIVE, pixels from previous frames can be kept around for up to 20 frames, and used as equal participants in the pull-push algorithm. When a block has changed significantly, a KILL_BLOCK signal is encoded for that block, and all pixels for that block from previous frames are discarded. For scene changes, a KILL_ALL_BLOCKS signal will discard all previous pixels from previous frames.

We flush the previous frame samples for a given block by using a special word (KILL_BLOCK) instead of encoding the sample. When this code is seen, the block that corresponds to the offset for that sample, will be marked, and all corresponding samples from previous frames are flushed. Additionally, we do not increment the pointer into the offset table, such that the next sample in the stream falls in the current block. We encode the KILL_BLOCK signals for new blocks in all the packets of a given frame. Currently, there exists a possibility of reusing samples from a wrong frame under few error scenarios; but this contition can be remedied by encoding a sequence number with the KILL_BLOCK signal (analogous to MPEG-2 slice id information).

Blocks that do not change will slowly improve in quality because they are reusing samples from previous frames; therefore we wish to add more samples to the blocks which are changing more rapidly and are not reusing samples. We accomplish this by inserting negative values in the SKIP TABLE in the following way. When a block is killed, we set its corresponding SKIP TABLE entry to a negative value (currently -10). After we have gone around once for all blocks in the image, we only visit blocks that have a negative SKIP TABLE entry and increment its SKIP TABLE for each sample received. This continues until there are no more negative SKIP TABLE entries left. This increases the reconstructed quality of blocks that are not reusing previous samples. This does not violate our globalness principle, since we still have at least one sample per every block if they fit in a packet.

5 Results

In this section we evaluate the performance of our compression system. Before we proceed it is important to note two caveats. First, the policies of the encoder will greatly determine the quality of the decompressed stream. The encoder can make many decisions. For example, it can make decisions about which blocks to flush or keep, what offset to start sending samples from, from which levels samples should be drawn, what proportion of luminance/chrominance samples to use, among other decisions. We have manually found reasonable settings for our video streams. In the optimal case, the encoder would make these decisions automatically. Secondly, we have used the signal-to-noise ratio metric (SNR) for evaluating our results. It is well known that SNR is not an optimal measurement for image quality. It is acceptable for comparing the algorithms

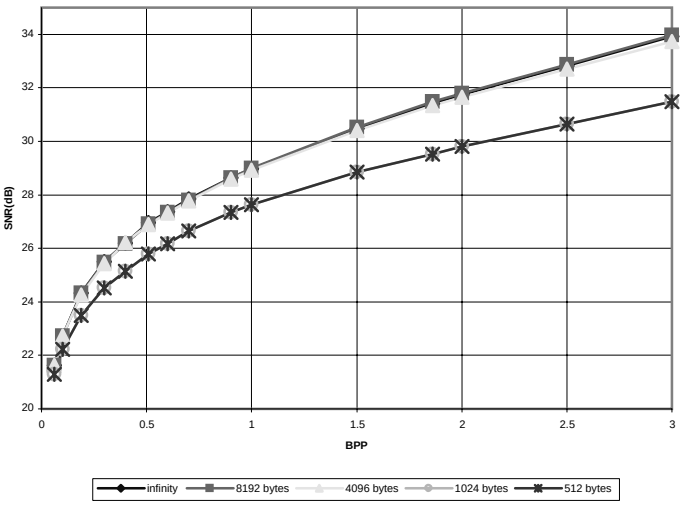


Figure 6: Rate-distortion curve on the grayscale 512x512 "Lena" test image.

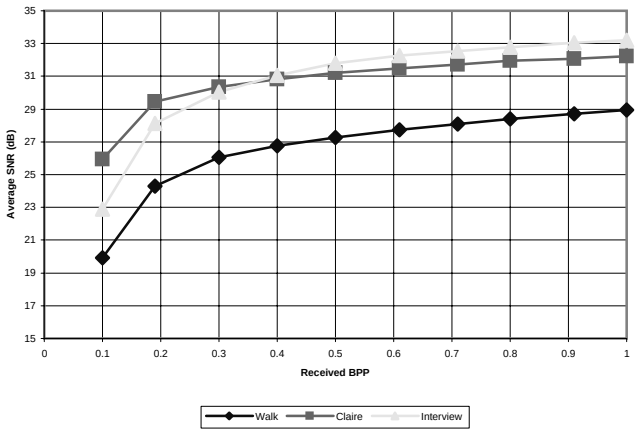


Figure 7: Average SNR of 3 color sequences with 100 frames encoded at 1 bpp [2] and decoded with different packet drop rates yielding different bpp. receive rates.

based on the same transform with different settings [17]. A better measurement would be based on models of the human visual system; but these are usually harder to implement or compute than the SNR.

Figure 6 shows the rate distortion curve for 512x512 grayscale image, compressed for different target bit per pixels (bpp) and different packet sizes. Large packet sizes are important for large images. If the packet is not larger than the number of blocks in an image, then there will not be enough space to go around all the blocks once, and more importantly, the algorithm will not make use of the SKIP TABLE, which allows it to get more samples in needed areas. The drawback of using large packets is that they are more likely to fragmented and lost. When a packet is fragmented, and one of its fragments get lost, the whole packet is lost. For small images, a packet size of 1024 bytes is adequate. For our experiments we used a packet size of 1024 bytes because it is compatible with the maximum packet size of most networks.

Figure 7 shows how the quality degrades gracefully for different kinds of video sequences. For these sequences, temporal and spacial locality has been used. The first sequence, *Walk*, contains a men in suits walking from a car, the scene has high detail and motion. The second sequence, *Claire* is a standard head and shoulders shot. Lastly, the *Interview*, consists of three scenes: a person walking into a room, a head and shoulders shot of the person talking inside the room, and close up of her face. All three sequences contain 100 frames, and were encoded at 1bpp. To generate all the data, the sequences were decoded with different packet drop rates calculating the average SNR of all frames. The packet drop rate determines the independent probability that a packet will be dropped. Over a whole sequence, a video encoded at 1bpp and decoded with a packet drop rate of 30%, will have a receive bpp of 0.7bpp. The slope of all three curves is very similar, showing that it degrades slowly regardless of the kind of video.

The algorithm handles bursty packet losses well. Figure 8 shows the frame by frame SNR for the 10 second *Interview* (320x240 color) sequence compressed at 0.33 bpp. This sequence is composed of three shots. The first 22 frames is a shot sequence of the person walking into an office. The stride of the person and camera angle makes the shot contain one slow motion frame and one fast motion frame, to give the resulting wave-like shape for the SNR during that shot. The second shot is a head and shoulders shot of the person being interview in her office. This shot lasts until frame 77. The last shot is a close up of the person. The quality of the image is above 30dB for most of the sequence, there is a short dip between frame 77 and frame 78, but it does not take long to recover.

Figure 9 shows the same sequence under bursty packet loss. The dashed line represents the actual bit rate during the reception of each frame. This figure shows that even under heavy loss (receiving less that 0.1 bpp), the quality does not degrade significantly. At the end of the first burst, in frame 28, the quality level recovers rapidly. Additionally, the quality hardly degrades during the second burst, between frames 37 and 47.

The complexity of the algorithm is simple enough to allow a software-only implementation. Table 1 shows the decoding frame rate for different sequences. The algorithm was run on a common Intel Pentium Pro 200Mhz processor running Linux and the X windows system. The frame rate is not very sensitive to the amount of data received. The decoding time is dominated by the pull-push algorithm after all the samples received from the network have been placed in the image. The color sequence ran at 50% lower frame rate, than the comparable grayscale sequence. This makes sense, since we have to reconstruct the chrominance data which is half the size of the luminance data for color sequences. Displaying QCIF sequences in real time would not be a problem, and with a faster machine and an efficient display system, the same might be possible for CIF sequences.

6 Conclusions

The NAIVE system that we have presented is an initial step towards a video compression system tailored specifically for computer networking environments. NAIVE satisfies our initial design goals. It supports broadcast over large-area network and maintains scalability. NAIVE is tolerant to packet loss at any point along the network

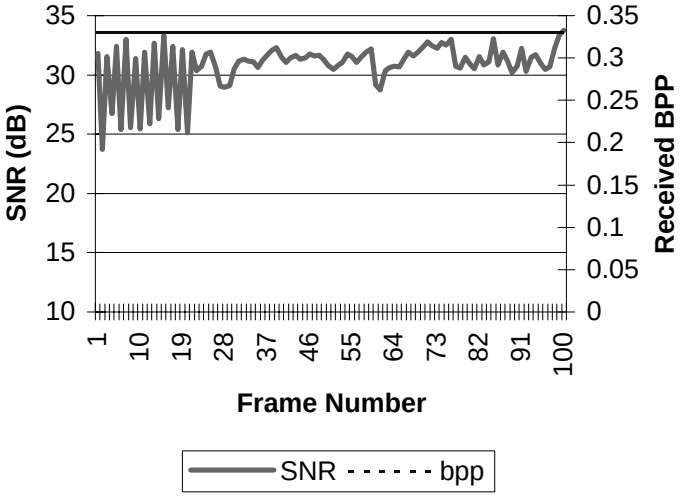


Figure 8: Base: SNR for each frame vs. the bpp received per frame, constant receive rate of 0.33 bpp

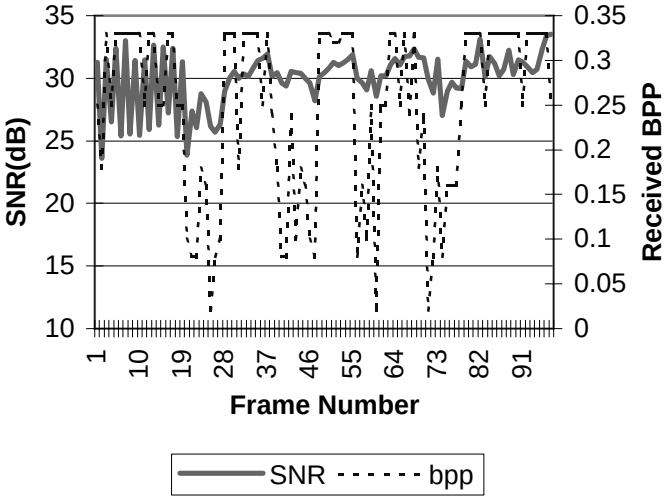


Figure 9: Bursty: SNR for each frame vs. the bpp received per frame, there are bursty errors, so the receive rate drops sporadically

Test Sequence	fps 1bpp	fps 0.5bpp
interview (color 320x240)	23.5	25.3
susie (gray 352x240)	34.81	36.32
qclaire (gray 176x144)	76.7	84.9

Table 1: Decoding frame rates (without displaying) for different sequences.

from the sender to the receiver. In fact, the intentional dropping of packets at the source is one method of increasing the effective compression of the bit stream. Similarly, the selective dropping of packets at the receiver effectively sheds CPU load. A NAIVE sender can also dynamically vary its transmission bandwidth when required by the video sequence in order to maintain a given quality level. In all cases, the receiver of a NAIVE video stream is able to reconstruct a reasonable approximation of an entire frame using a minimum of information (i.e. a single packet). The reception of additional packets further enhances the quality of the frame. Finally, our system degrades gracefully under severe packet losses.

Fundamentally, the randomizing of samples used in our NAIVE method has the effect of decorrelating the input signal and effective compression methods essentially depend on highly correlated input signals. Thus, our NAIVE algorithm sacrifices compression ratio, as compared to other video compression techniques, in order to achieve our design goals. We believe that other compression techniques can be layered onto our NAIVE methods to achieve substantially improved compression. For instance, differential encoding methods could be applied to all samples in a packet following the initial sample. Variable length encoding techniques can be applied within individual packets to reduce redundancy in the transmitted symbols. We are also hopeful that motion compensation techniques can be applied within our framework by encoding motion vector for each block. These motion vectors would imply that a block of samples in all pyramid levels would be copied to the current block. Thus, the sender would make no specific assumption concerning which samples are available at the receiver, only that those samples within the transferred block would form the best basis for reconstructing the desired block. It is also possible to incorporate embedded coding techniques to the samples within each packet. This would potentially allow for trading off the quantization of samples for increased sampling density.

Another shortcoming of our NAIVE method is that the sender is fundamentally unable to make any quality guarantees to any particular receiver. The need for such a guarantee might arise based from an economics driven approach where particular receivers pay a premium for assurances of a given quality level. Layering is an effective technique for satisfying such requirements. We believe that our NAIVE method could be extended to provide layering. Finally, we plan to integrate audio into our framework in the near future. We'll either adapt the NAIVE mechanisms to audio or use one of the standard protocols for audio distribution.

In summary, we view our NAIVE algorithm as starting point for the development of a new class of video compression methods that are well suited for computer networks. By considering the realities of real networks we believe that is possible to define new classes of algorithms that are scalable in broadcast applications and degrade gracefully under variations in network activity.

Acknowledgements

We would like to thank Aaron Isaksen for his help in preparing our videos. Support for this research was provided by DARPA contract N30602-97-1-0283, and Massachusetts Institute of Technology's Laboratory for Computer Science.

References

[1] H.263: Video coding for low bitrate communication. *Draft ITU-T Recommendation H.263.*, May 1996.

[2] Elen Amir, Steven McCanne, and Martin Vetterli. A layered dct coder for internet video. In *IEEE International Conference on Image Processing*, pages 13–16, Lausanne, Switzerland, September 1996.

[3] E. Asbun and E. Delp. Real-time error concealment in compressed digital video streams. *Proceedings of the Picture Coding Symposium 1999*, April 1999.

[4] Ernst W. Biersack. A performance study of foward error correction in atm networks. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSDAV) 1993*, pages 391–399, Heildelberg, Germany, November 1993.

[5] G. Bishop, H. Fuchs, L. McMillan, and E. Scher Zaiger. Frameless rendering: Double buffering considered harmful. *Computer Graphics (SIGGRAPH 94)*, pages 175–176, 1994.

[6] Jean-Chrysostome Bolot, Hugues Crepin, and Andres Vega Garcia. Analysis of audio packet loss in the internet. In *NOSDAV*, pages 154–165, Durham, NH, 1995.

[7] Jean-Chrysostome Bolot, Thierry Turletti, and Ian Wakeman. Scalable feedback control for multicast video distribution in the internet. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1994*, pages 58–67, London, UK, 1994.

[8] J. Boyce and R. Gaglianello. Packet loss effects on mpeg video sent over the public internet. *ACM Multimedia, 1998*, 1998.

[9] P. Burt and E. Adelson. Laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31(4), April 1983.

[10] P. J. Burt. Moment images, polynomial fit filters, and the problem of surface interpolation. In *Proceedings of Computer Vision and Pattern Recognition*, pages 144–152. IEEE Computer Society Press, June 1988.

[11] Y. Chung, J. Kim, and C. Kuo. Dct based error concealment for rtsp video over a modem internet connection. *International Symposium on Circuits and Systems '98*, May 1998.

[12] I. Cidon, A. Khamisy, and M. Sidi. Analysis of packet loss processes in high-speed networks. *IEEE Trans. Info. Theory*, 39(1), January 1993.

[13] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1990*, September 1990.

[14] A. A. El-Gamal and T. M. Cover. Achievable rates for multiple descriptions. *IEEE Trans. Information Theory*, 28:851–857, 1982.

[15] S. Gortler, R. Grzeszczuk, and M. Cohen R. Szeliski. The lumigraph. *Computer Graphics (SIGGRAPH 96)*, pages 43–54, 1996.

[16] V. K. Goyal, J. Kovacevic, R. Arean, and M. Vetterli. Multiple description transform coding of images. *Proc. IEEE Int. Conf. Image Processing*, October 1998.

[17] Yung-Kai Lai, Jin Li, and C.-C. Jay Kuo. A wavelet approach to compressed image quality measurement. *30th Annual Asilomar Conference on Signals, Systems, and Computers*, November 1996.

[18] A. Lippman and W.Butera. Coding image sequences for interactive retrieval. *ACM: CACM*, 32(7):852–860, July 1989.

[19] Peter Litwinowicz and Lance Williams. Animating images with drawings. In *Computer Graphics (SIGGRAPH 94)*, pages 409–412, 1994.

[20] S. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE PAMI*, 11, July 1989.

[21] Steven R. McCanne. *Scalable Video Coding and Transmission for Internet Multicast Video*. PhD thesis, University of California, Berkeley, December 1996.

[22] D. P. Mitchell. Generating antialiased images at low sampling densities. *Computer Graphics (SIGGRAPH'87)*, 21(4):65–72, July 1987.

[23] M. T. Orchard, Y. Wang, V. Vaishampayan, and A. R. Reibman. Redundancy rate-distortion analysis of multiple description coding using pairwise correlating transforms. *Proc. IEEE Int. Conf. Image Processing*, October 1997.

[24] J. OŔourke. *Computational Geometry in C*. Cambridge University Press, 1993.

[25] Sassan Pejhan, Mischa Schwartz, and Dimitris Anastassiou. Error control using retransmission schemes in multicast transport protocols for real-time media. *IEEE/ACM Transactions on Networking*, 4(3):413–427, June 1996.

[26] M. Podolsky, C. Romer, and S. Mccanne. Simulation of fec-based error control for packet audio on the internet. *INFOCOM 98*, March 1998.

[27] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[28] Injong Rhee. Error control techniques for interactive low-bit rate video transmission over the internet. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM) 1998*, pages 290–301, Vancouver, B.C., 1998.

[29] S. Servetto, K. Ramchandran, V. Vaishampayan, and K. Nahrstedt. Multiple description wavelet based image coding. *In the Proceedings of the IEEE International Conference on Image Processing (ICIP)*, October 1998.

[30] D. Terzopoulos. Regularization of inverse visual problems involving discontinuities. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(4):413–424, July 1986.

[31] S Tsekeridou, I Pitas, and C LeBuhan. An error concealment scheme for mpeg-2 coded video sequences. *ISCAS '97*, pages 1289–1292, June 1997.

[32] P. H. Westerink, J. H. Weber, and J. W. Limpers. Adaptive channel error protection of subband encoded images. *IEEE Transactions on Communications*, 41(3):454–459, March 1993.

[33] X. Rex Xu, Andrew C. Myers, Hui Zhang, and Raj Yavatkar. Resilient multicast support for contininuous-media applications. In *International Workshop on Network and Operating System Support for Digital Audio and Video (NOSDAV) 1997*, pages 183–193, St. Louis, MO, May 1997.

[34] W. Zeng and B Liu. Geometric structure based directional filtering for error concealment in image video transmission. *SPIE vol 2601, Wireless Data Transmission, Photonics East '95*, October 1995.

Appendix C:

Image-Based Visual Hulls

Wojciech Matusik*
Laboratory for Computer Science
Massachusetts Institute of Technology

Chris Buehler*
Laboratory for Computer Science
Massachusetts Institute of Technology

Ramesh Raskar‡
Department of Computer Science
University of North Carolina - Chapel Hill

Steven J. Gortler†
Division of Engineering and Applied Sciences
Harvard University

Leonard McMillan*
Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

In this paper, we describe an efficient image-based approach to computing and shading visual hulls from silhouette image data. Our algorithm takes advantage of epipolar geometry and incremental computation to achieve a constant rendering cost per rendered pixel. It does not suffer from the computation complexity, limited resolution, or quantization artifacts of previous volumetric approaches. We demonstrate the use of this algorithm in a real-time virtualized reality application running off a small number of video streams.

Keywords: Computer Vision, Image-Based Rendering, Constructive Solid Geometry, Misc. Rendering Algorithms.

1 Introduction

Visualizing and navigating within virtual environments composed of both real and synthetic objects has been a long-standing goal of computer graphics. The term "Virtualized Reality™", as popularized by Kanade [23], describes a setting where a real-world scene is "captured" by a collection of cameras and then viewed through a virtual camera, as if the scene was a synthetic computer graphics environment. In practice, this goal has been difficult to achieve. Previous attempts have employed a wide range of computer vision algorithms to extract an explicit geometric model of the desired scene.

Unfortunately, many computer vision algorithms (e.g. stereo vision, optical flow, and shape from shading) are too slow for real-time use. Consequently, most virtualized reality systems employ off-line post-processing of acquired video sequences. Furthermore, many computer vision algorithms make unrealistic simplifying assumptions (e.g. all surfaces are diffuse) or impose impractical restrictions (e.g. objects must have sufficient non-periodic textures) for robust operation. We present a new algorithm for synthesizing virtual renderings of real-world scenes in real time. Not only is our technique fast, it also makes few simplifying assumptions and has few restrictions.

*(wojciech | cbuehler | mcmillan)@graphics.lcs.mit.edu
†sjg@cs.harvard.edu
‡raskar@cs.unc.edu

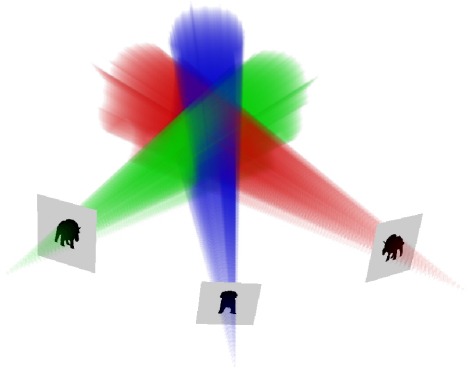


Figure 1 - The intersection of silhouette cones defines an approximate geometric representation of an object called the visual hull. A visual hull has several desirable properties: it contains the actual object, and it has consistent silhouettes.

Our algorithm is based on an approximate geometric representation of the depicted scene known as the visual hull (see Figure 1). A visual hull is constructed by using the visible silhouette information from a series of reference images to determine a conservative shell that progressively encloses the actual object. Based on the principle of *calculatus eliminatus* [28], the visual hull in some sense carves away regions of space where the object "is not".

The visual hull representation can be constructed by a series of 3D constructive solid geometry (CSG) intersections. Previous robust implementations of this algorithm have used fully enumerated volumetric representations or octrees. These methods typically have large memory requirements and thus, tend to be restricted to low-resolution representations.

In this paper, we show that one can efficiently render the exact visual hull without constructing an auxiliary geometric or volumetric representation. The algorithm we describe is "image based" in that all steps of the rendering process are computed in "image space" coordinates of the reference images.

We also use the reference images as textures when shading the visual hull. To determine reference images that can be used, we compute which reference cameras have an unoccluded view of each point on the visual hull. We present an image-based visibility algorithm based on epipolar geometry and McMillan's occlusion compatible ordering [18] that allows us to shade the visual hull in roughly constant time per output pixel.

Using our *image-based visual hull* (IBVH) algorithm, we have created a system that processes live video streams and renders the observed scene from a virtual camera's viewpoint in real time. The resulting representation can also be combined with traditional computer graphics objects.

2 Background and Previous Work

Kanade's virtualized reality system [20] [23] [13] is perhaps closest in spirit to the rendering system that we envision. Their initial implementations have used a collection of cameras in conjunction with multi-baseline stereo techniques to extract models of dynamic scenes. These methods require significant off-line processing, but they are exploring special-purpose hardware for this task. Recently, they have begun exploring volume-carving methods, which are closer to the approach that we use [26] [30].

Pollard's and Hayes' [21] immersive video objects allow rendering of real-time scenes by morphing live video streams to simulate three-dimensional camera motion. Their representation also uses silhouettes, but in a different manner. They match silhouette edges across pairs of views, and use these correspondences to compute morphs to novel views. This approach has some limitations, since silhouette edges are generally not consistent between views.

Visual Hull. Many researchers have used silhouette information to distinguish regions of 3D space where an object is and is not present [22] [8] [19]. The ultimate result of this carving is a shape called the object's *visual hull* [14]. A visual hull always contains the object. Moreover, it is an equal or tighter fit than the object's convex hull. Our algorithm computes a view-dependent, sampled version of an object's visual hull each rendered frame.

Suppose that some original 3D object is viewed from a set of reference views R. Each reference view r has the silhouette s_r with interior pixels covered by the object. For view r one creates the cone-like volume vh_r defined by all the rays starting at the image's point of view p_r and passing through these interior points on its image plane. It is guaranteed that the actual object must be contained in vh_r. This statement is true for all r; thus, the object must be contained in the volume $vh_R=\bigcap_{r\in R} vh_r$. As the size of R goes to infinity, and includes all possible views, vh_R converges to a shape known as the visual hull vh_∞ of the original geometry. The visual hull is not guaranteed to be the same as the original object since concave surface regions can never be distinguished using silhouette information alone.

In practice, one must construct approximate visual hulls using only a finite number of views. Given the set of views R, the approximation vh_R is the best conservative geometric description that one can achieve based on silhouette information alone (see Figure 1). If a conservative estimate is not required, then alternative representations are achievable by fitting higher order surface approximations to the observed data [2].

Volume Carving. Computing high-resolution visual hulls can be tricky matter. The intersection of the volumes vh_r requires some form of CSG. If the silhouettes are described with a polygonal mesh, then the CSG can be done using polyhedral CSG, but this is very hard to do in a robust manner.

A more common method used to convert silhouette contours into visual hulls is volume carving [22] [8] [29] [19] [5] [27]. This method removes unoccupied regions from an explicit volumetric representation. All voxels falling outside of the projected silhouette cone of a given view are eliminated from the volume. This process is repeated for each reference image. The resulting volume is a quantized representation of the visual hull according to the given volumetric grid. A major advantage of our view-dependent method is that it minimizes artifacts resulting from this quantization.

CSG Rendering. A number of algorithms have been developed for the fast rendering of CSG models, but most are ill suited for our task. The algorithm described by Rappoport [24], requires that each solid be first decomposed to a union of convex primitives. This decomposition can prove expensive for complicated silhouettes. Similarly, the algorithm described in [11] requires a rendering pass for each layer of depth complexity. Our method does not require preprocessing the silhouette cones. In fact, there is no explicit data structure used to represent the silhouette volumes other than the reference images.

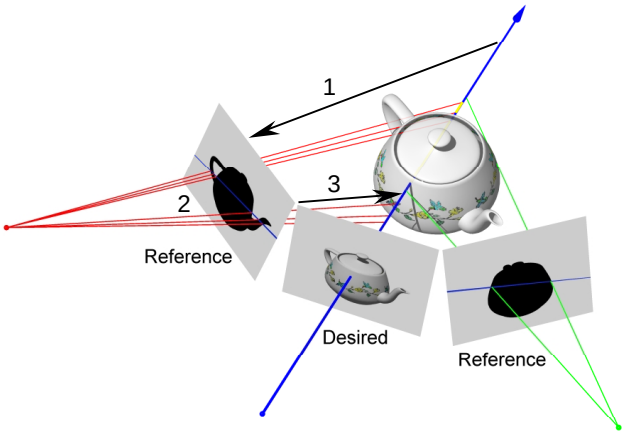


Figure 2 – Computing the IBVH involves three steps. First, the desired ray is projected onto a reference image. Next, the intervals where the projected ray crosses the silhouette are determined. Finally, these intervals are lifted back onto the desired ray where they can be intersected with intervals from other reference images.

Using ray tracing, one can render an object defined by a tree of CSG operations without explicitly computing the resulting solid [25]. This is done by considering each ray independently and computing the interval along the ray occupied by each object. The CSG operations can then be applied in 1D over the sets of intervals. This approach requires computing a 3D ray-solid intersection. In our system, the solids in question are a special class of cone-like shapes with a constant cross section in projection. This special form allows us to compute the equivalent of 3D ray intersections in 2D using the reference images.

Image-Based Rendering. Many different image-based rendering techniques have been proposed in recent years [3] [4] [15] [6] [12]. One advantage of image-based rendering techniques is their stunning realism, which is largely derived from the acquired images they use. However, a common limitation of these methods is an inability to model dynamic scenes. This is mainly due to data acquisition difficulties and preprocessing requirements. Our system generates image-based models in real-time, using the same images to construct the IBHV and to shade the final rendering.

3 Visual-Hull Computation

Our approach to computing the visual hull has two distinct characteristics: it is computed in the image space of the reference images and the resulting representation is viewpoint dependent. The advantage of performing geometric computations in image space is that it eliminates the resampling and quantization artifacts that plague volumetric approaches. We limit our sampling to the pixels of the desired image, resulting in a view-dependent visual-hull representation. In fact, our IBVH representation is equivalent to computing exact 3D silhouette cone intersections and rendering the result with traditional rendering methods.

Our technique for computing the visual hull is analogous to finding CSG intersections using a ray-casting approach [25].

Given a desired view, we compute each viewing ray's intersection with the visual hull. Since computing a visual hull involves only intersection operations, we can perform the CSG calculations in any order. Furthermore, in the visual hull context, every CSG primitive is a generalized cone (a projective extrusion of a 2D image silhouette). Because the cone has a fixed (scaled) cross section, the 3D ray intersections can be reduced to cheaper 2D ray intersections. As shown in Figure 2 we perform the following steps: 1) We project a 3D viewing ray into a reference image. 2) We perform the intersection of the projected ray with the 2D silhouette. These intersections result in a list of intervals along the ray that are interior to the cone's cross-section. 3) Each interval is then lifted back into 3D using a simple projective mapping, and then intersected with the results of the ray-cone intersections from other reference images. A naïve algorithm for computing these IBVH ray intersections follows:

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    for each scanline s in d
      for each pixel p in s
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        intervals int2D = calcIntervals(l2,r.silEdges)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

To analyze the efficiency of this algorithm, let n be the number of pixels in a scanline. The number of pixels in the image d is $O(n^2)$. Let k be the number of reference images. Then, the above algorithm has an asymptotic running time $O(ikn^2)$, where i is the time complexity of the `calcIntervals` routine. If we test for the intersection of each projected ray with each of the e edges of the silhouette, the running time of `calcIntervals` is $O(e)$. Given that l is the average number of times that a projected ray intersects the silhouette[1], the number of silhouette edges will be $O(ln)$. Thus, the running time of `IBVHisect` to compute all of the 2D intersections for a desired view is $O(lkn^3)$.

The performance of this naïve algorithm can be improved by taking advantage of incremental computations that are enabled by the epipolar geometry relating the reference and desired images. These improvements will allow us to reduce the amortized cost of 1D ray intersections to $O(l)$ per desired pixel, resulting in an implementation of `IBVHisect` that takes $O(lkn^2)$.

Given two camera views, a reference view r and a desired view d, we consider the set of planes that share the line connecting the cameras' centers. These planes are called *epipolar planes*. Each epipolar plane projects to a line in each of the two images, called an *epipolar line*. In each image, all such lines intersect at a common point, called the *epipole*, which is the projection of one of the camera's center onto the other camera's view plane [9].

As a scanline of the desired view is traversed, each pixel projects to an epipolar line segment in r. These line segments emanate from the epipole e_{dr}, the image of d's center of projection onto r's image plane (see Figure 3), and trace out a "pencil" of epipolar lines in r. The slopes of these epipolar line segments will either increase or decrease monotonically depending on the direction of traversal (Green arc in Figure 3). We take advantage of this monotonicity to compute silhouette intersections for the whole scanline incrementally.

[1] We assume reference images also have $O(n^2)$ pixels.

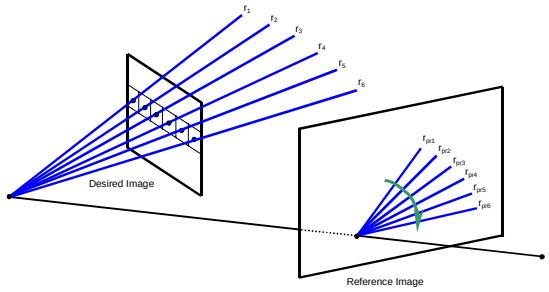


Figure 3 – The pixels of a scanline in the desired image trace out a pencil of line segments in the reference image. An ordered traversal of the scanline will sweep out these segments such that their slope about the epipole varies monotonically.

The silhouette contour of each reference view is represented as a list of edges enclosing the silhouette's boundary pixels. These edges are generated using a 2D variant of the marching cubes approach [16]. Next, we sort the $O(nl)$ contour vertices in increasing order by the slope of the line connecting each vertex to the epipole. These sorted vertex slopes divide the reference image domain into $O(nl)$ bins. Bin B_i has an extent spanning between the slopes of the ith and $i+1$st vertex in the sorted list. In each bin B_i we place all edges that are intersected by epipolar lines with a slope falling within the bin's extent[2]. During `IBVHisect` as we traverse the pixels along a scanline in the desired view, the projected corresponding view rays fan across the epipolar pencil in the reference view with either increasing or decreasing slope. Concurrently, we step through the list of bins. The appropriate bin for each epipolar line is found and it is intersected with the edges in that bin. This procedure is analogous to merging two sorted lists, which can be done in a time proportional to the length of the lists ($O(nl)$ in our case).

For each scanline in the desired image we evaluate n viewing rays. For each viewing ray we compute its intersection with edges in a single bin. Each bin contains on average $O(l)$ silhouette edges. Thus, this step takes $O(l)$ time per ray. Simultaneously we traverse the sorted set of $O(nl)$ bins as we traverse the scanline. Therefore, one scanline is computed in $O(nl)$ time. Over n scanlines of the desired image, and over k reference images, this gives a running time of $O(lkn^2)$. Pseudocode for the improved algorithm follows.

```
IBVHisect (intervalImage &d, refImList R){
  for each referenceImage r in R
    computeSilhouetteEdges (r)
  for each pixel p in desiredImage d do
    p.intervals =  {0..inf}
  for each referenceImage r in R
    bins b = constructBins(r.caminfo, r.silEdges, d.caminfo)
    for each scanline s in d
      incDec order = traversalOrder(r.caminfo,d.caminfo,s)
      resetBinPositon(b)
      for each pixel p in s according to order
        ray3D  ry3 = compute3Dray(p,d.camInfo)
        lineSegment2D l2 = project3Dray(ry3,r.camInfo)
        slope m = ComputeSlope(l2,r.caminfo,d.caminfo)
        updateBinPosition(b,m)
        intervals int2D = calcIntervals(l2,b.currentbin)
        intervals int3D = liftIntervals(int2D,r.camInfo,ry3)
        p.intervals = p.intervals ISECT int3D
}
```

[2] Sorting the contour vertices takes $O(nl \log(nl))$ and binning takes $O(nl^2)$. Sorting and binning over k reference views takes $O(knl \log(nl))$ and $O(knl^2)$ correspondingly. In our setting, $l << n$ so we view this preprocessing stage as negligible.

It is tempting to apply further optimizations to take greater advantage of epipolar constraints. In particular, one might consider rectifying each reference image with the desired image prior to the ray-silhouette intersections. This would eliminate the need to sort, bin, and traverse the silhouette edge lists. However, a call to `liftInterval` would still be required for each pixel, giving the same asymptotic performance as the algorithm presented. The disadvantage of rectification is the artifacts introduced by the two resampling stages that it requires. The first resampling is applied to the reference silhouette to map it to the rectified frame. The second is needed to unrectify the computed intervals of the desired view. In the typical stereo case, the artifacts of rectification are minimal because of the closeness of the cameras and the similarity of their pose. But, when computing visual hulls the reference cameras are positioned more freely. In fact, it is not unreasonable for the epipole of a reference camera to fall within the field of view of the desired camera. In such a configuration, rectification is degenerate.

4 Visual-Hull Shading

The IBVH is shaded using the reference images as textures. In order to capture as many view-dependent effects as possible a view-dependent texturing strategy is used. At each pixel, the reference-image textures are ranked from "best" to "worst" according to the angle between the desired viewing ray and rays to each of the reference images from the closest visual hull point along the desired ray. We prefer those reference views with the smallest angle [7]. However, we must avoid texturing surface points with an image whose line-of-sight is blocked by some other point on the visual hull, regardless of how well aligned that view might be to the desired line-of-sight. Therefore, visibility must be considered during the shading process.

When the visibility of an object is determined using its visual hull instead of its actual geometry, the resulting test is conservative– erring on the side of declaring potentially visible points as non-visible. We compute visibility using the visual hull, VH_R, as determined by IBVHisect. This visual hull is represented as intervals along rays of the desired image *d*. Pseudocode for our shading algorithm is given below.

```
IBVHshade(intervalImage &d, refImList R){
  for each pixel p in d do
    p.best = BIGNUM
  for each referenceImage r in R do
    for each pixel p in d do
       ray3D ry3 = compute3Dray(p,d.camInfo)
       point3 pt3 = front(p.intervals,ry3)
       double s  = angleSimilarity(pt3,ry3,r.camInfo)
       if isVisible(pt3,r,d)
         if (s < p.best)
           point2 pt2 = project(pt3,r.camInfo)
           p.color = sample_color(pt2,r)
           p.best = s
}
```

The `front` procedure finds the front most geometric point of the IBVH seen along the ray. The `IBVHshade` algorithm has time complexity $O(vkn^2)$, where *v* is the cost for computing visibility of a pixel.

Once more we can take advantage of the epipolar geometry in order to incrementally determine the visibility of points on the visual hull. This reduces the amortized cost of computing visibility to $O(l)$ per desired pixel, thus giving an implementation of `IBVHshade` that takes $O(lkn^2)$.

Consider the visibility problem in flatland as shown in Figure 4. For a pixel *p*, we wish to determine if the front-most point on the visual hull is occluded with respect to a particular reference image by any other pixel interval in *d*.

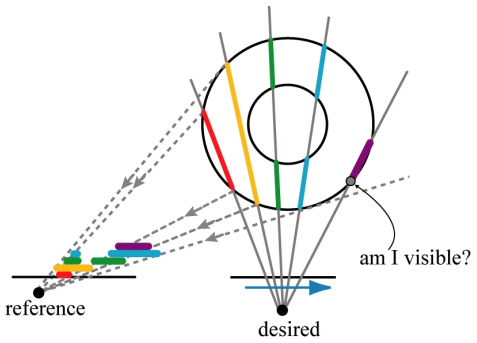


Figure 4 – In order to compute the visibility of an IBVH sample with respect to a given reference image, a series of IBVH intervals are projected back onto the reference image in an occlusion-compatible order. The front-most point of the interval is visible if it lies outside of the unions of all preceding intervals.

Efficient calculation can proceed as follows. For each reference view *r*, we traverse the desired-view pixels in front-to-back order with respect to *r* (left-to-right in Figure 4). During traversal, we accumulate coverage intervals by projecting the IBVH pixel intervals into the reference view, and forming their union. For each front most point, `pt3`, we check to see if its projection in the reference view is already covered by the coverage intervals computed thus far. If it is covered, then `pt3` is occluded from *r* by the IBVH. Otherwise, `pt3` is not occluded from *r* by either the IBVH or the actual (unknown) geometry.

```
visibility2D(intervalFlatlandImage &d, referenceImage r){
  intervals  coverage = <empty>
  for each pixel p in d do \\front to back in r
    ray2D ry2 = compute2Dray(p,d.camInfo)
    point2 pt2 = front(p.intervals,ry2);
    point1D p1 = project(pt2,r.camInfo)
    if contained(p1,coverage)
      p.visible[r] = false
    else
      p.visible[r] = true
    intervals tmp  =
                prjctIntrvls(p.intervals,ry2,r.camInfo)
    coverage = coverage UNION tmp
}
```

This algorithm runs in $O(nl)$, since each pixel is visited once, and containment test and unions can be computed in $O(l)$ time.

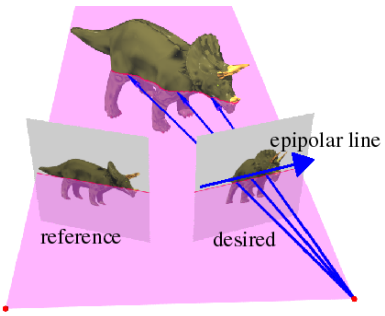


Figure 5 – Ideally, the visibility of points in 3D could be computed by applying the 2D algorithm along epipolar planes.

In the continuous case, 3D visibility calculations can be reduced to a set of 2D calculations within epipolar planes (Figure 5), since all visibility interactions occur within such planes. However, the extension of the discrete 2D algorithm to a complete discrete 3D solution is not trivial, as most of the discrete pixels in our images do not exactly share epipolar planes. Consequently, one must be careful in implementing conservative 3D visibility.

First, we consider each of the intervals stored in d as a solid frustum with square cross section. To determine visibility of a (square) pixel p correctly we consider $S_{p,}$ the set of all possible epipolar planes which touch p. There are at least two possible definitions for whether p is visible: (1) p is visible along **all** planes in S_p , (2) p is visible along **any** plane in $S_{p.}$ Clearly the first definition results in more pixels that are labeled not visible, therefore, it is better suited when using a large number of reference images. With a small number of reference images, the second definition is preferred. Implementing efficient exact algorithms for these visibility definitions is difficult, therefore, we use conservative algorithms; if the pixel is truly invisible we never label it as visible. However, the algorithms could label some pixel as invisible though it is in fact visible.

An algorithm that conservatively computes visibility according to the first definition is performed as follows. We define an epipolar wedge starting from the epipole e_{rd} in the desired view extending out to a one pixel-width interval on the image boundary. Depending on the relative camera views, we traverse the wedge either toward or away from the epipole [17]. For each pixel in this wedge, we compute visibility with respect to the pixels traversed earlier in the wedge using the 2D visibility algorithm. If a pixel is computed as *visible* then no geometry within the wedge could have occluded it in the reference view. We use a set of wedges whose union covers the whole image. A pixel may be touched by more than one wedge, in these cases its final visibility is computed as the *AND* of the results obtained from each wedge.

The algorithm for the second visibility definition works as follows. We do not consider all possible epipolar lines that touch pixel p but only some subset of them such that at least one line touches each pixel. One such subset is all the epipolar lines that pass through the centers of the image boundary pixels. This particular subset completely covers all the pixels in the desired image; denser subsets can also be chosen. The algorithm computes `visibility2D` for all epipolar lines in the subset. Visibility for a pixel might be computed more than once (e.g., the pixels near the epipole are traversed more often). We *OR* all obtained visibility results. Since we compute `visibility2D` for up to $4n$ epipolar lines in k reference images the total time complexity of this algorithm is $O(lkn^2)$. In our real-time system we use small number of reference images (typically four). Thus, we use the algorithm for the second definition of visibility.

The total time complexity of our IBVH algorithms is $O(lkn^2)$, which allows for efficient rendering of IBVH objects. These algorithms are well suited to distributed and parallel implementations. We have demonstrated this efficiency with a system that computes IBVHs in real time from live video sequences.

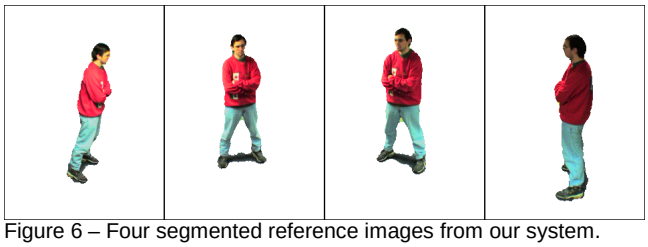


Figure 6 – Four segmented reference images from our system.

5 System Implementation

Our system uses four calibrated Sony DFW500 FireWire video cameras. We distribute the computation across five computers, four that process video and one that assembles the IBVH (see Figure 6). Each camera is attached to a 600 MHz desktop PC that captures the video frames and performs the following processing steps. First, it corrects for radial lens distortion using a lookup table. Then it segments out the foreground object using background-subtraction [1] [10]. Finally, the silhouette and texture information are compressed and sent over a 100Mb/s network to a central server for IBVH processing.

Our server is a quad-processor 550 MHz PC. We interleave the incoming frame information between the 4 processors to increase throughput. The server runs the IBVH intersection and shading algorithms. The resulting IBVH objects can be depth-buffer composited with an OpenGL background to produce a full scene. In the examples shown, a model of our graphics lab made with the Canoma modeling system was used as a background.

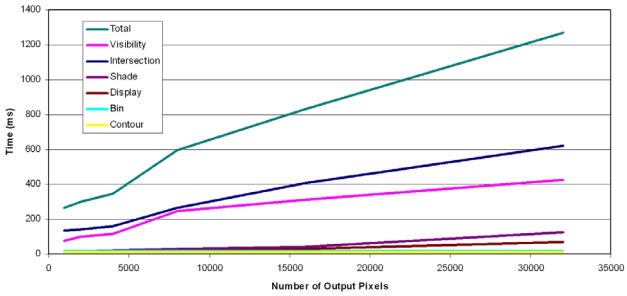


Figure 7 – A plot of the execution times for each step of the IBVH rendering algorithm on a single CPU. A typical IBVH might cover approximately 8000 pixels in a 640×480 image and it would execute at greater than 8 frames per second on our 4 CPU machine.

In Figure 7, the performances of the different stages in the IBVH algorithm are given. For these tests, 4 input images with resolutions of 256×256 were used. The average number of times that a projected ray crosses a silhouette is 6.5. Foreground segmentation (done on client) takes about 85 ms. We adjusted the field of view of the desired camera, to vary the number of pixels occupied by the object. This graph demonstrates the linear growth of our algorithm with respect to the number of output pixels.

6 Conclusions and Future Work

We have described a new image-based visual-hull rendering algorithm and a real-time system that uses it. The algorithm is efficient from both theoretical and practical standpoints, and the resulting system delivers promising results.

The choice of the visual hull for representing scene elements has some limitations. In general, the visual hull of an object does not match the object's exact geometry. In particular, it cannot represent concave surface regions. This shortcoming is often considered fatal when an accurate geometric model is the ultimate goal. In our applications, the visual hull is used largely as an imposter surface onto which textures are mapped. As such, the visual hull provides a useful model whose combination of accurate silhouettes and textures provides surprisingly effective renderings that are difficult to distinguish from a more exact model. Our system also requires accurate segmentations of each image into foreground and background elements. Methods for accomplishing such segmentations include chromakeying and image differencing. These techniques are subject to variations in cameras, lighting, and background materials.

We plan to investigate techniques for blending between textures to produce smoother transitions. Although we get impressive results using just 4 cameras, we plan to scale our system up to larger numbers of cameras. Much of the algorithm parallelizes in a straightforward manner. With k computers, we expect to achieve $O(n^2 l \log k)$ time using a binary-tree based structure.

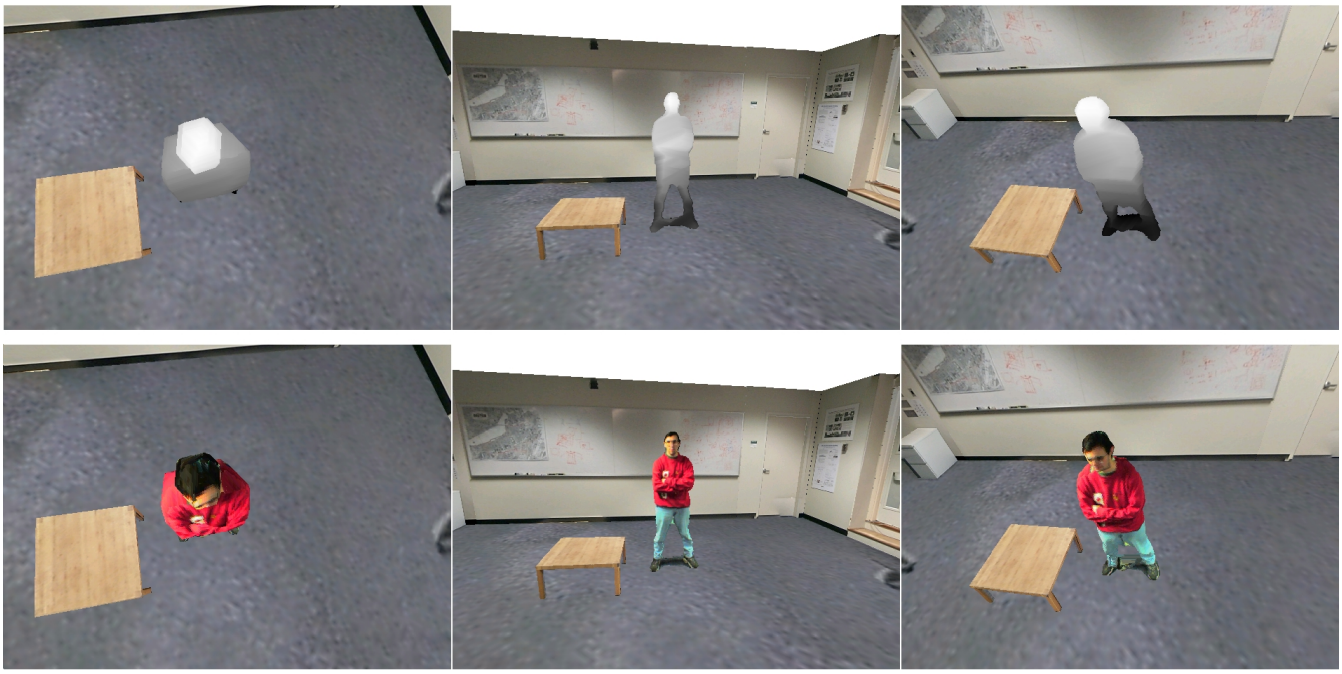


Figure 8 - Example IBVH images. The upper images show depth maps of the computed visual hulls. The lower images show shaded renderings from the same viewpoint. The hull segment connecting the two legs results from a segmentation error caused by a shadow.

7 Acknowledgements

We would like to thank Kari Anne Kjølaas, Annie Choi, Tom Buehler, and Ramy Sadek for their help with this project. We also thank DARPA and Intel for supporting this research effort. NSF Infrastructure and NSF CAREER grants provided further aid.

8 References

[1] Bichsel, M. "Segmenting Simply Connected Moving Objects in a Static Scene." *IEEE PAMI* 16, 11 (November 1994), 1138-1142.

[2] Boyer, E., and M. Berger. "3D Surface Reconstruction Using Occluding Contours." *IJCV* 22, 3 (1997), 219-233.

[3] Chen, S. E. and L. Williams. "View Interpolation for Image Synthesis." *SIGGRAPH 93*, 279-288.

[4] Chen, S. E. "Quicktime VR – An Image-Based Approach to Virtual Environment Navigation." *SIGGRAPH 95*, 29-38.

[5] Curless, B., and M. Levoy. "A Volumetric Method for Building Complex Models from Range Images." *SIGGRAPH 96*, 303-312.

[6] Debevec, P., C. Taylor, and J. Malik, "Modeling and Rendering Architecture from Photographs." *SIGGRAPH 96*, 11-20.

[7] Debevec, P.E., Y. Yu, and G. D. Borshukov, "Efficient View-Dependent Image-based Rendering with Projective Texture Mapping." *Proc. of EGRW* 1998 (June 1998).

[8] Debevec, P. *Modeling and Rendering Architecture from Photographs.* Ph.D. Thesis, University of California at Berkeley, Computer Science Division, Berkeley, CA, 1996.

[9] Faugeras, O. *Three-dimensional Computer Vision: A Geometric Viewpoint.* MIT Press, 1993.

[10] Friedman, N. and S. Russel. "Image Segmentation in Video Sequences." *Proc 13th Conference on Uncertainty in Artifical Intelligence* (1997).

[11] Goldfeather, J., J. Hultquist, and H. Fuchs. "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System." *SIGGRAPH 86*, 107-116.

[12] Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen. "The Lumigraph." *SIGGRAPH 96*, 43-54.

[13] Kanade, T., P. W. Rander, and P. J. Narayanan. "Virtualized Reality: Constructing Virtual Worlds from Real Scenes." *IEEE Multimedia* 4, 1 (March 1997), 34-47.

[14] Laurentini, A. "The Visual Hull Concept for Silhouette Based Image Understanding." *IEEE PAMI* 16,2 (1994), 150-162.

[15] Levoy, M. and P. Hanrahan. "Light Field Rendering." *SIGGRAPH 96*, 31-42.

[16] Lorensen, W.E., and H. E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm." *SIGGRAPH 87*, 163-169.

[17] McMillan, L., and G. Bishop. "Plenoptic Modeling: An Image-Based Rendering System." *SIGGRAPH 95*, 39-46.

[18] McMillan, L. *An Image-Based Approach to Three-Dimensional Computer Graphics*, Ph.D. Thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 1997.

[19] Moezzi, S., D.Y. Kuramura, and R. Jain. "Reality Modeling and Visualization from Multiple Video Sequences." *IEEE CG&A* 16, 6 (November 1996), 58-63.

[20] Narayanan, P., P. Rander, and T. Kanade. "Constructing Virtual Worlds using Dense Stereo." *Proc. ICCV* 1998, 3-10.

[21] Pollard, S. and S. Hayes. "View Synthesis by Edge Transfer with Applications to the Generation of Immersive Video Objects." *Proc. of VRST*, November 1998, 91-98.

[22] Potmesil, M. "Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images." *CVGIP* 40 (1987), 1-29.

[23] Rander, P. W., P. J. Narayanan and T. Kanade, "Virtualized Reality: Constructing Time Varying Virtual Worlds from Real World Events." *Proc. IEEE Visualization* 1997, 277-552.

[24] Rappoport, A., and S. Spitz. "Interactive Boolean Operations for Conceptual Design of 3D solids." *SIGGRAPH 97*, 269-278.

[25] Roth, S. D. "Ray Casting for Modeling Solids." *Computer Graphics and Image Processing*, 18 (February 1982), 109-144.

[26] Saito, H. and T. Kanade. "Shape Reconstruction in Projective Grid Space from a Large Number of Images." *Proc. of CVPR*, (1999).

[27] Seitz, S. and C. R. Dyer. "Photorealistic Scene Reconstruction by Voxel Coloring." *Proc. of CVPR* (1997), 1067-1073.

[28] Seuss, D. "The Cat in the Hat," *CBS Television Special* (1971).

[29] Szeliski, R. "Rapid Octree Construction from Image Sequences." *CVGIP: Image Understanding* 58, 1 (July 1993), 23-32.

[30] Vedula, S., P. Rander, H. Saito, and T. Kanade. "Modeling, Combining, and Rendering Dynamic Real-World Events from Image Sequences." *Proc. 4th Intl. Conf. on Virtual Systems and Multimedia* (Nov 1998).

Appendix D:

Dynamically Reparameterized Light Fields

Aaron Isaksen
MIT LCS Computer Graphics Group
aisaksen@graphics.lcs.mit.edu
http://graphics.lcs.mit.edu/~aisaksen

Leonard McMillan
MIT LCS Computer Graphics Group
mcmillan@graphics.lcs.mit.edu
http://graphics.lcs.mit.edu/~mcmillan

Steven J. Gortler
Harvard University
sjg@cs.harvard.edu
http://www.cs.harvard.edu/~sjg

Abstract

An exciting new area in computer graphics is the synthesis of novel images with photographic effect from an initial database of reference images. This is the primary theme of image-based rendering algorithms. This research extends the light field and lumigraph image-based rendering methods and greatly extends their utility, especially in scenes with much depth variation. First, we have added the ability to vary the apparent focus within a light field using intuitive camera-like controls such as a variable aperture and focus ring. As with lumigraphs, we allow for more general and flexible focal surfaces than a typical focal plane. However, this parameterization works independently of scene geometry; we do not need to recover actual or approximate geometry of the scene for focusing. In addition, we present a method for using multiple focal surfaces in a single image rendering process.

Introduction

The light field [Levoy96] and lumigraph [Gortler96] rendering methods use similar four-dimensional data structures for representing a half-space of rays through a volume of space. We will refer to this data structure as a *ray database*. Novel images are synthesized from this database by querying it for each ray needed to construct a desired view. The set of viewpoints that can be generated are restricted to those within an empty region of space lying outside of the convex hull of objects in the scene that are composed entirely of rays from the selected half-space. Several ray databases can be used to represent a scene, and desired images may combine rays queried from different databases. A pair of planes is typically used to parameterize a ray database, although other parameterizations have been suggested [Camahort96].

A continuous representation of a ray database would be sufficient for generating any desired viewpoint under the previously described viewing restrictions. However, continuous databases are impractical or unattainable for all but the most trivial cases. In practice, we must work with a discretely sampled ray databases. As with any sampled representation the issues of choosing an appropriate initial sampling density as well as defining methods for reconstructing continuous representations from the given sample set are crucial factors in representing the underlying model. A previous motivation for selecting a new parameterization for the ray database was to facilitate better or more uniform sampling. Improving the uniformity of sampling density helps find an adequate sample rate to avoid aliasing artifacts. These artifacts due to an initial undersampling cannot be removed though a subsequent process unless additional information or constraints are provided.

The choice of a ray database parameterization also affects the choice of reconstruction methods that can be used in synthesizing desired views. Thus, even when supplied with an adequately sampled dataset, it is frequently the case that a non-ideal reconstruction filter will introduce artifacts into the result, whereas a better reconstruction filter on the same dataset might have generated a more correct result. This process of introducing artifacts in the reconstruction process is often called *postaliasing* [Mitchell88]. Postaliasing artifacts include excessive high-frequency leakage, sometimes called ringing, and excessive pass-band attenuation, or blurring. The standard planar parameterizations of ray databases have a substantial impact on the choice of reconstruction filters. Here, we present an alternative parameterization that allows for a more flexible choice of reconstruction filters.

To date, most light fields are constructed for object-centered, or outside-looking-in, environments rather than viewer-centered, or inside-looking-out environments. This is not entirely coincidental: the original two-plane light field can best represent points that are located near the exit plane of the ray database. Objects located a small distance from this exit plane will appear out of focus (either blurred or ghosted, depending on the extent of aperture filtering). Thus, an object-centered model is better suited, as the distance the object lies from the exit plane is well represented by a plane.

We would like to represent inside-looking-out light fields with a wide variations in depth. This requires a more flexible parameterization of the ray database.

In the sections that follow, we present an extension of the light field parameterization that introduces the notion of a focal surface. Then, we discuss how the treatment of a light field as a discrete synthetic aperture camera will provide dynamic variations of depth of field. Next, we explain how moving and orienting the focal surface will affect the images created using this ray database. We then present the idea of using multiple focal planes, how to create them, and how to use them when rendering. Finally, we present ideas on how one would optimally make these multiple focal planes.

Focal-Plane Abstraction

Overview

Our parameterization of ray databases is analogous to a two-dimensional array of pinhole cameras treated as a single optical system with a synthetic aperture. Each constituent pinhole camera captures an image in clear focus, and this camera array acts as a discrete aperture in the image formation process. Because we have a discrete, finite aperture, some amount of depth of field defocusing will be present in our renderings. However, by using an arbitrary plane of focus, we can establish correspondences between the rays from different pinhole cameras. That is, we can control which items we want to be in focus. This is essentially the approach used to simulate depth-of-field effects in ray-traced images [Cook84].

Mathematical formulation

In the standard two-plane ray database parameterization there is an entrance plane, with parameters *(s, t)* and an exit plane with parameters *(u, v)*, where each ray is uniquely determined by the 4-tuple *(s, t, u, v)*, as illustrated in Figure 1. It is often instructive to consider and/or interpret subspaces of such a ray database [Gu96]. A two dimensional subspace given by fixed *s* and *t* values resembles an image, whereas fixed *u* and *v* values give a hypothetical radiance function. Fixing *t* and *v* gives rise to an epipolar plane image, or EPI [Bolles87].

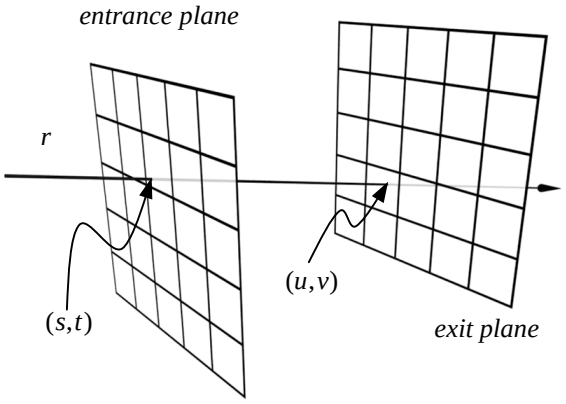


Figure 1: In the standard light field parameterization, a ray is referenced by its intersections with an entrance plane and an exit plane.

Our new parameterization is best described in terms of three 2-D surfaces, which are shown in Figure 2 below. Our *camera surface*, described in terms of two parameters *s* and *t*, is identical in function to the entrance plane of the standard parameterization. Our *image surfaces* describe a discrete set of rays from a given point, (*s*,*t*), on the camera surface and has the form ($u_{s,t}$, $v_{s,t}$). The elements of the ray database are accessed via a four-tuple *(s, t,* $u_{s,t}$, $v_{s,t}$*)*. Our focal surface is described in terms of two parameters, (u_F, v_F), that are independent of all others. Our parameterization also requires a mapping $M_F : (s,t,u_F,v_F) \rightarrow (u_{s,t},v_{s,t})$; this maps from focal surface parameters to image surface parameters given a specific camera surface coordinate. That is, this mapping tells us which ray *(s,t,*$u_{s,t}$,$v_{s,t}$*)* in the ray database is the same as the ray (s,t,u_F,v_F).

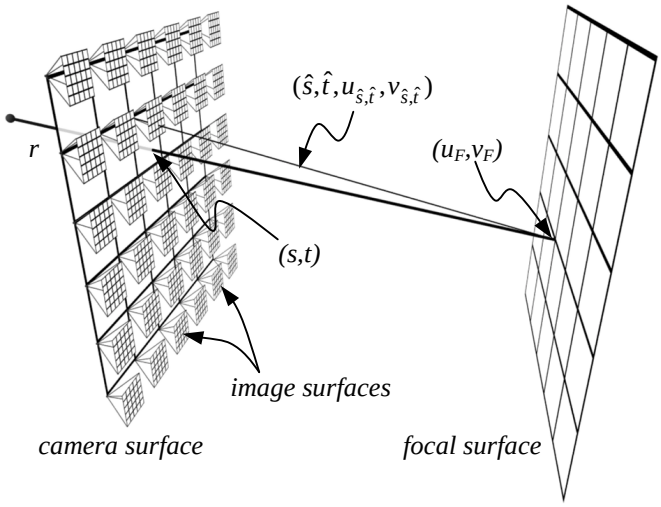


Figure 2: Our new parameterization has added a focal surface.

When querying a discrete ray database, we wish to find the ray $\hat{r}$, a ray that has been recorded in the ray database, that best approximates a ray *r*. Given *r* and a focal surface *F*, one calculates the ray's intersections with the camera surface and focal surface to get (s,t,u_F,v_F). Then, (s,t,u_F,v_F) is quantized to $(\hat{s},\hat{t},u_F,v_F)$, because the camera surface is sampled discretly, not continuously. This 4-tuple is passed throught the mapping $M_F : (\hat{s},\hat{t},u_F,v_F) \rightarrow (u_{\hat{s},\hat{t}},v_{\hat{s},\hat{t}})$ to obtain the nearest ray $\hat{r}$ in the ray database which passes through $(\hat{s},\hat{t})$. By varying the focal surface and the focal surface mapping, different rays $\hat{r}$ will be returned for a given ray *r*.

The key difference between our ray-database query formulation and that used by the light field and the lumigraph is the identification of independent image and focal surfaces. In the case of a standard two-plane parameterization the mapping function from *(*u_F,v_F*)* to *(*$u_{s,t}$,$v_{s,t}$*)* is an identity. Therefore, every point on the camera surface shares a common image surface, and the image surface is coincident with the focal surface.

However, we have separated these surfaces, and the relationship between them is determined by the focal-plane-to-image-plane mapping function. This mapping can be modified dynamically, and these alterations do not effect the organization of the underlying ray database. Thus, we defer the selection of the focal surface until image synthesis time, and make the specification of this surface available to the user.

The addition of a focal surface abstraction has only a minor impact on the image synthesis process. Assume that the center-of-projection of the desired image lies at the origin. In the case where the camera, image, and focal surfaces are defined as planes the mapping from a desired ray to a database query can be structured as a pair of projective mappings.

$$\begin{bmatrix} rs \\ rt \\ r \end{bmatrix} = \mathbf{C}\bar{d}$$

$$\begin{bmatrix} wu \\ wv \\ w \end{bmatrix} = \mathbf{I}_{\hat{s},\hat{t}}\mathbf{F}\bar{d}$$

In these equations, $\bar{d}$ is the direction of the desired ray. The 3 by 3 matrix **C** gives the intersection of the ray with the camera plane in terms of *s* and *t*. Likewise, the matrix **F** gives the (u_F,v_F) intersection of the ray with the focal plane. The 3 by 3 matrix $\mathbf{I}_{s,t}$ maps focal-plane coordinates into pixel coordinates of the specified camera, $(\hat{s},\hat{t})$. These mappings are computed for each image synthesized. In light fields and lumigraphs, all points on the camera plane share a common image plane that is also the focal plane. Therefore, only two mapping functions are required, one for the entrance plane and one for the exit plane. In our parameterization the focal surface is defined dynamically. Thus, **F** is determined by the user. The composite map, $\mathbf{I}_{\hat{s},\hat{t}}\mathbf{F}$, must be determined for every discrete position on the camera plane where an image plane is specified, for example at 256 points. This quantity can be computed lazily, but in any case it is only a small overhead compared to the ray queries.

Discrete Synthetic Aperture Camera

When quantizing $(s,t) \rightarrow (\hat{s},\hat{t})$, the nearest ray $\hat{r}$ is constrained to pass through $(\hat{s},\hat{t})$. Clearly, except in the case where $(s,t) = (\hat{s},\hat{t})$, the ray $\hat{r}$ is not the same as *r*, and errors in the output image are apparent. Evaluating M_F only once for each ray *r* leads to noticeable discontinuities in the output image, as the quantization $(s,t) \rightarrow (\hat{s},\hat{t})$ suddenly jumps to a new location on the camera surface, even when (s,t) may only change a small amount (see figure 3 below).

To reduce these discontinuities, one can ask for the four rays from the four nearest $(\hat{s},\hat{t})$ samples. Then one can interpolate between these rays to find a better approximation than any one of these rays alone. In the original light-field paper, this was referred to as *st-interpolation*. However, a more general approach would be to take a linear combination of a set of nearby rays. This is analogous to a camera system's point-spread function. Unfortunately, this trades discontinuities for focusing problems, as we have now added a finite aperture and therefore a limited depth of field (related to the distances between the samples on the camera surface). Nevertheless, we are used to dealing with real world camera systems which exhibit these depth of field problems: we accept them, and even derive artistic value from them. However, we are not accepting of discontinuities in an image, and the tradeoff is a useful one.

Figure 3: Although the image is clearly focused, using only a single nearest neighbor ray creates noticeable discontinuities.

Variable apertures

Depth of Field

One can render depth of field effects by blending a larger set of approximate rays. If M_F is evaluated for all cameras $(\hat{s},\hat{t})$ within a given radial distance from (s,t), the aperture radius of the synthetic camera is increased. In Figure 4, 7 different cameras will be used for a single input ray. Whereas one can only *decrease* the aperture radius by sampling the camera surface more densely, one can *increase* the aperture at run-time by averaging more rays together (i.e. changing the radius of the gray circle in Figure 4).

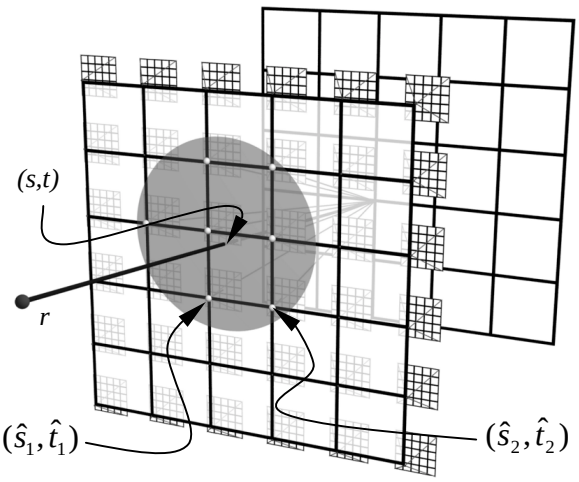


Figure 4: By including rays from all cameras within a radius of the actual camera surface intersection, we can increase the aperture of our synthetic camera.

By combining rays from cameras farther away from (s,t), only objects that are near the focal surface will be in focus. By definition, the set of nearest rays obtained through M_F for a

given (u_F, v_F) will intersect at (u_F, v_F), regardless of $(\hat{s},\hat{t})$. That is, these rays are 'looking' at the point (u_F, v_F) on the focal surface *F*. If there was an object at that location when the ray database was captured, the rays will agree on the color of that surface (up to view dependent variations). In the left side of Figure 5, r_1, r_2, r_3, and r_4 agree. However, as the actual object gets farther from the focal surface, the agreement of the rays diverge, as in the right side of Figure 5. By increasing the aperture radius, more rays will be averaged, and these colors will diverge faster. Thus, we have a control over depth of field that is intuitive to photographers: the f-stop on a camera is inversely proportional to aperture radius.

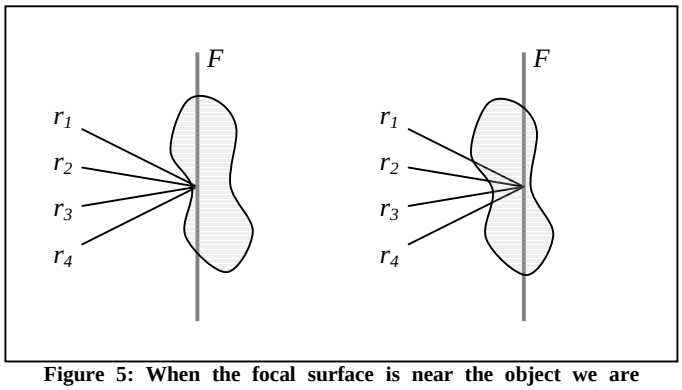


Figure 5: When the focal surface is near the object we are looking at, the rays agree and the object appears to be in focus (left). If the object is further from the focal surface (right), then the rays do not agree, and the object will appear out of focus.

Seeing through Objects

Other algorithms could be used to create effects not available to photographers. In Figure 6, we used an aperture that included every camera in our data set. If that were taken with a single real camera, the aperture would be about the size of a 3-story building! Because our depth of field is so narrow, we can "look through" objects.

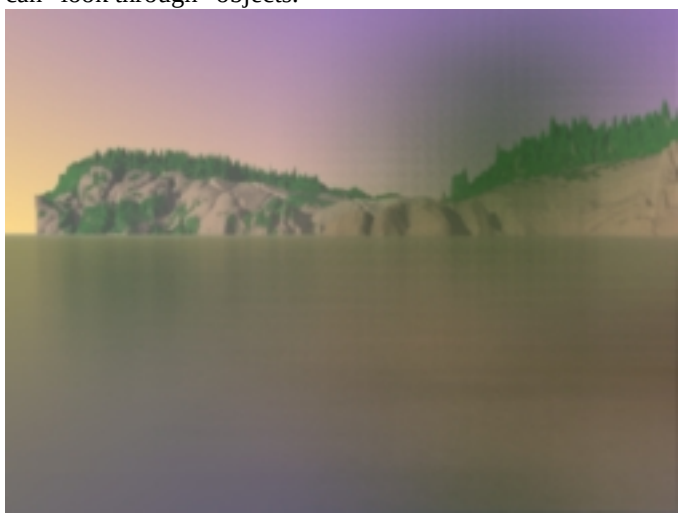


Figure 6: By making the aperture very large, we are able to look through objects. In this case, there is a tree and island occluding the hills where the slight haze appears. Figures 8,9, and 10 are the same scene with a smaller aperture; the tree is visible.

Varying focus

Though we have shown a way to change the size of the aperture, this could have been done with the standard light field parameterization. We will now show what can be accomplished with a parameterization that allows one to *dynamically control* what is in focus.

Moving the Focal Surface

A photographer using a camera can not only change the depth of field, but he can change what is in focus. Using our parameterization, one changes the focal surface in order to change what appears in focus. As before, a ray *r*, a camera surface, and a focal surface *F* intersect at (s,t,u_F,v_F). This 4-tuple is then quantized and passed through a mapping $M_F : (\hat{s},\hat{t},u_F,v_F) \rightarrow (u_{\hat{s},\hat{t}}, v_{\hat{s},\hat{t}})$ to obtain the nearest ray $\hat{r}$ in the ray database.

When the focal surface is changed to *F'*, the same ray *r* now intersects the camera and focal surfaces at the new coordinates $(\hat{s},\hat{t},u_{F'},v_{F'})$. Thus, by dynamically changing the focal surface, we are dynamically changing which ray $\hat{r}$ in the ray database is 'nearest' to the ray *r*. When we change the focal surface from F to F', we are changing from $\hat{r}$, a ray that passes through (u_F, v_F), to $\hat{r}$, a ray that passes through $(u_{F'}, v_{F'})$ (see Figure 7). Since objects nearest to the focal surface intersection will be in focus when using a finite aperture, we have added a variable focus into the light field parameterization.

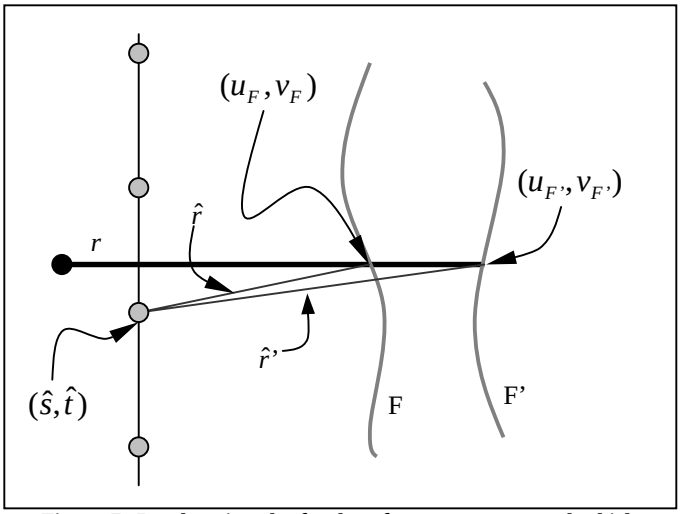


Figure 7: By changing the focal surface, we can control which ray in the ray database best approximates a given ray *r*.

Without the focal surface mapping, the light field always returned the same ray $\hat{r}$ for an input ray *r*. Since this deficency is tied into the storage of the ray database, light fields and lumigraphs have a fixed focus. Whereas the standard light fields implementations could render either Figure 8 *or* Figure 9, it could not do both without rerendering the database, clearly not a dynamic operation. Depth-

corrected lumigraphs would allow a dynamic focal surface, but only a single one.

Freely oriented focal surfaces

Since the focal surface determines which regions of space in the light field will be in focus, moving the focal surface allows the user to focus on different parts of the scene. For example, when we use a plane parallel to the image plane as a focal surface, it makes it easy to see what lies at a given depth in the scene. In figure 8, we have chosen to make the tree in focus, while figure 9 focuses on the hills behind the island. Moving the focal plane only changes 1) the mapping function M_F and 2) where the ray r intersects with F at (u_F, v_F). This is a simple change that does not affect the storage of the ray database.

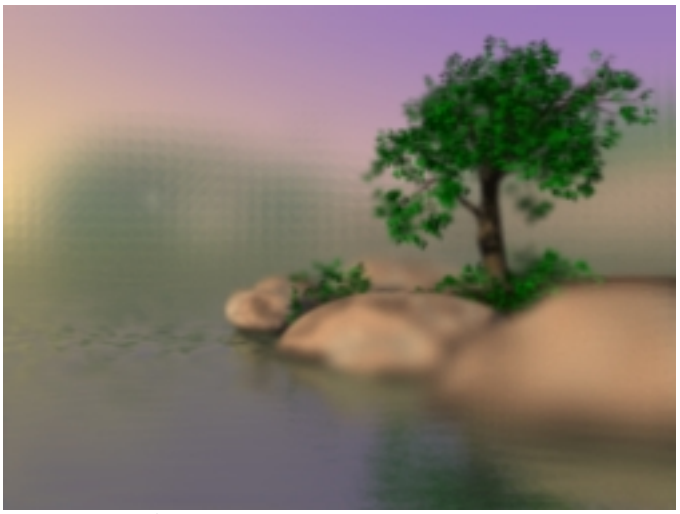


Figure 8: A focal plane has been placed through the tree.

Figure 9: The same scene as Figure 8, but with the focal plane passing through the hills behind the island.

We do not need to keep the focal surface parallel to the image plane. If we orient the plane such that it passes through various objects in the scene, we can constrain these objects to be in focus. In Figure 10, we pass the focal surface through the front rock, part of the tree, and the rock at the left edge of the island. This non-parallel focal plane is available to photographers that use a bellows on their camera, but bellows are not common equipment and can be difficult to align with the optical system. And, of course the focus cannot be dynamically changed after film has been exposed. Since rotating the focal plane again simply causes a change of M_F and a different (u_F, v_F) for a ray r, it no more difficult to arbitrarily orient a focal plane than it is to move one.

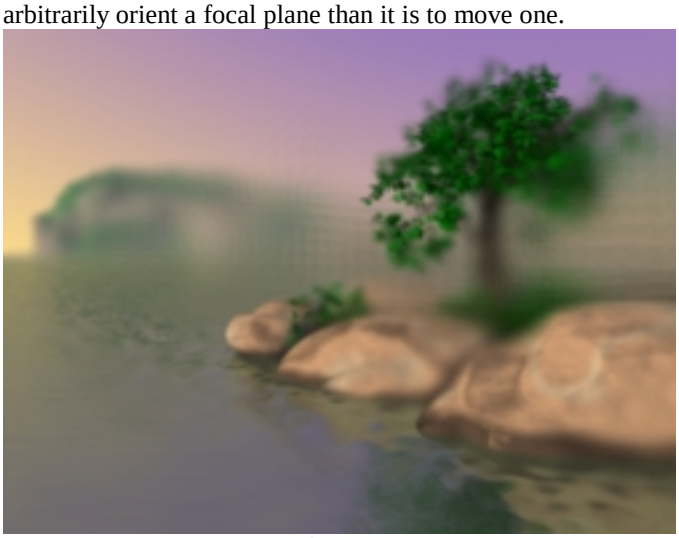


Figure 10: We have placed a focal plane that is not parallel to the image plane. In this case, the plane passes through part of the tree, the front rock, and the leftmost rock on the island. The plane of focus can be seen intersecting with the water in a line.

Non-planar focal surfaces

Clearly, these example scenes can not be entirely focused with a single plane. Of course, the focal surfaces do not have to be planar. One could create a focal surface out of a parameterized surface patch that passes through key points in a scene. Or, one could even use a depth map of the scene as a focal surface, insuring that all visible surfaces were in focus. This would analogous to depth-corrected lumigraphs, where a proxy surface helps focus the representation But, in reality, these depth maps would be hard and/or expensive to obtain with simple hardware, and would likely only be applicable to synthetic ray databases.

Multiple Focal Surfaces

In general, we would like to have more than just the points near a single surface in clear focus. One solution is to use multiple focal surface, something not available to real cameras. In a real lens system, only one continuous region is in focus at one time. However, since we are not confined by physical optics, we can have two or more distinct regions that are in focus. For example, in Figure 11, the red bull in front and the monitors in back are in focus, yet the objects in between, such as the yellow fish and the blue animal, are out of focus. Using a real camera, this can be done by first taking a set of pictures with different planes of focus, and then taking the best parts of each image and compositing them together [Haeberli94].

Figure 11: Using two focal surfaces allows us to make the front and back objects in focus, while those in the middle are blurry.

Since a ray *r* will intersect each focal surface, some scoring scheme is needed to pick which focal surface will be used. We would like to pick the focal surface which will make the picture look most focused, which means we need to pick the focal surface which is closest to the actual object being looked at. We can augment each focal surface with some scoring $\sigma(u_F, v_F)$, which is the likelihood a visible object is near (u_F, v_F). Then, we calculate σ for each focal surface, and we can pick the focal point with the best score σ. In Figure 12, we would like σ_2 to have the best score, for it is closest to the object. Note that an individual score $\sigma(u_F, v_F)$ is independent of the view direction; however, the set of scores compared for a particular ray *r* is dependent on the view direction. Therefore, although our scoring data can be developed with out view dependence, we can still extract view dependent information from it.

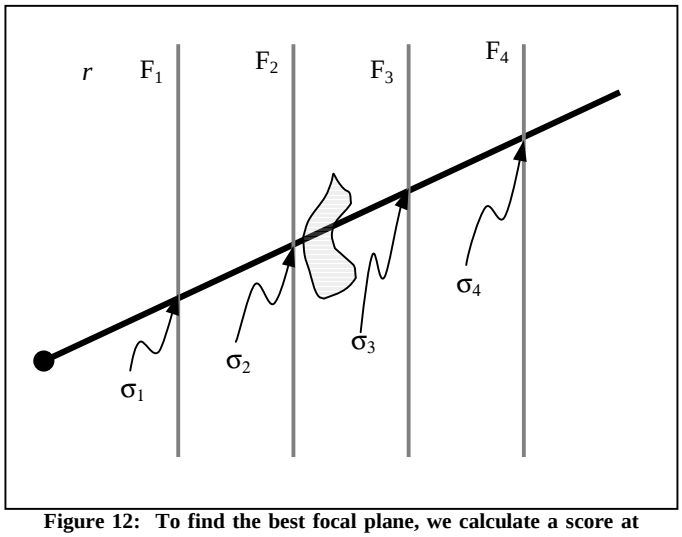


Figure 12: To find the best focal plane, we calculate a score at each intersection and take the focal plane with the best score. If the scoring system is good, the best score will be the one nearest the surface we are looking at.

If we are given the light-fields but no knowledge about the geometry of the scene, we must create these scores from information in the images alone. We would like to avoid computer vision techniques that involve deriving correspondences to discover the actual geometry of the scene, as vision algorithms may deal with ambiguities that are not relevant to generating synthetic images. For example, flat regions without texture can be troublesome to a vision system, and can make it hard to find an exact depth. However, when making images from a light field, picking the wrong depth near the same flat region would not affect us, because the region would still look in focus. Therefore, we would like a scoring system that allows us to take advantage of this extra freedom. And, because we only have the original images as input, we need a scoring system that can be easily created from these input images.

So, we have chosen to look for locations on the focal surfaces that approximate radiance functions. Whereas we have usually thought of the light-fields as looking in at objects, we can also use them to generate radiance functions. The collection of rays in the ray database that intersect at (u_F, v_F) is the discretized radiance function of the point (u_F, v_F). If the point lies on an object and is not occluded by other objects, the radiance function will be smooth, as in the left side of Figure 13 below. However, if the point is not near an actual object, then the radiance function will not be smooth, as in the right side of Figure 13.

To measure smoothness, we look for a lack of high frequencies in the radiance function. High frequencies in a radiance function identify 1) a point on a extremely specular surface, 2) an occlusions in the space between a point and a camera, or 3) a point in empty space. Thus, if we identify the regions where the there are no high frequencies in the radiance function, we know the point must be near a surface. Because of their high frequency content, we may miss areas that are actually on a surface but have an occluder in the way.

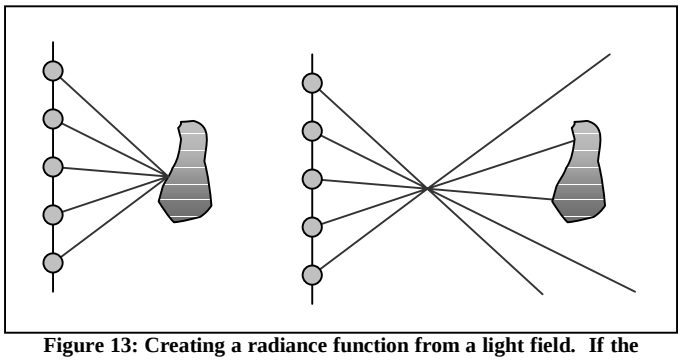


Figure 13: Creating a radiance function from a light field. If the radiance function is near an object, then the function will be smooth. If the radiance function is not near an object, it will vary greatly.

Because calculating the radiance function is a slow process, we create the scores for discrete points on each focal surface as a preprocessing step. This allows us to use expensive algorithms to setup the scores on our focal surfaces. Then, when we are rendering, we only need to recall the

prerecorded scores for each focal surface intersection and compare them.

To find the best focal surface for a ray, an algorithm must first obtain intersections and scores for each focal surface. Since this is linear in the number of focal surfaces, we would like to keep the number of focal surfaces small. However, the radiance functions are highly local, and small changes in the focal surface position can give large changes in the score. Nevertheless, the accuracy needed in placing the focal surfaces is not very high. That is, we do not need to find the exact surface that makes the object we are looking at in perfect focus; we just need to find a surface that is close to the object. Therefore, we can first calculate the scores by sampling the radiance functions for a large number of planes, and then 'squash' the scores down into a smaller set of planes using some function $\lambda(\sigma_i,...,\sigma_{i+m})$. For example, in Figure 14, we first calculate the radiance scores for 16 planes. Then, using some combining function λ, we combine the scores from these 16 planes down to new scores on four planes. These four planes and their scores will be used as focal surfaces at run time. We chose to use the maximizing function, that is, $\lambda(\sigma_i,...,\sigma_{i+m}) = \max(\sigma_i,...,\sigma_{i+m})$. Other non-linear or linear weighting function might provide better results. Figure 22 was created using 8 focal planes combined down to 4.

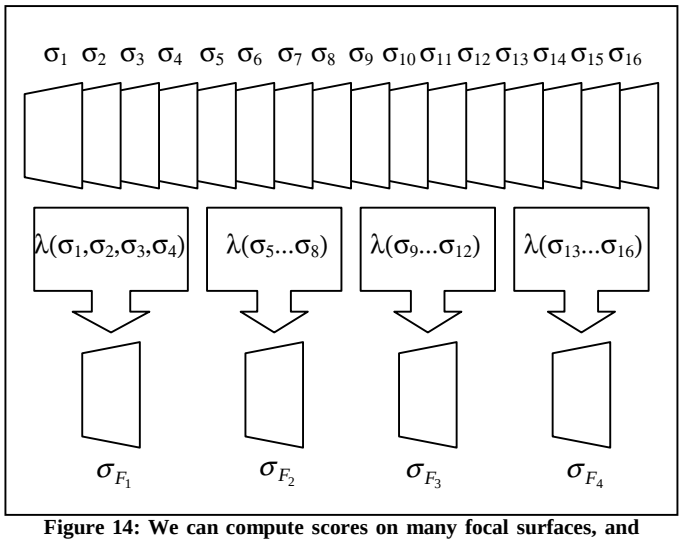


Figure 14: We can compute scores on many focal surfaces, and then combine them to a smaller set of focal surfaces, so the run-time algorithm will have less scores to compare.

Figure 15: Here is a visualization of the scores used on the front focal plane for the picture in Figures 11 and 20. The closer to white, the better the score σ, which means objects are likely to be located near that plane.

Figure 16: Here is a visualization of the scores on the back focal plane, analogous to Figure 15.

Selecting a Focal Surface through Auto-focus Techniques

We often select focal planes by hand, allowing a user to select the subject of the image. However, it is possible to determine these focal planes algorithmically, much an auto-focus camera. It is possible that by adapting these algorithms, we could identify a minimal set of focal planes that would put the most items in focus.

For a single plane, this would be analogous to using an auto-focus camera [Pentland87]. To do auto-focusing, one can create a series of images with an extremely narrow depth of field, where each image would put the focal plane at a different depth. This narrow depth of field can be implemented by increasing the aperture radius so that a ray from every camera is averaged to produce a single output ray. The resulting images will have out-of-focus and in-focus regions. The out-of-focus regions will have little high-

frequency energy, where as the regions in focus will. Since only structure very near the focal planes will be in focus, we know that the in-focus regions identify regions where there is structure. Thus, if we pass a high-pass filter over these narrow depth of field images and then identify the regions with high-frequency energy, we will find the regions in space where structure exists. Then, we can use the plane (or set of planes) which gives rise to the image with the most high-frequency energy: this plane is our auto-focus plane. Likewise, we could take the *n*-best planes for a multiple focal plane rendering.

Objects with little high-frequencies, even when they are in focus, such as flat regions with little texture, will not be detected by this process. However, objects with little high-frequency will look as good if they are out of focus as if they are in focus. Whereas using computer vision techniques to find depth from a set of images would have to further analyze these ambiguous regions, we do not have to delve further since several values will be good enough: we can simply take the best one that we find.

Results

The two light field data sets shown in this picture were created as follows. The tree data set, with 256 input images, was rendered in Povray 3.1, each image at 320x240. The stuffed animal data set was more complex to create. We attached an Electrim EDC1000E CCD camera (654x496) with an 16mm variable aperture lens to an X-Y motion platform from Arrick Robotics (30”x30” displacement). Then, we took 256 pictures on a (approximately) 16”x16” grid, which took approximately 30 minutes. To calibrate the camera, we first used a Faro Arm, a submillimeter accurate contact digitizer, to measure the 3-D spatial coordinates (x,y,z) of the centers of 24 large circular calibration pattern on two perpendicular planes filling the camera’s field of view. Then, using a picture of the calibration pattern, we found the centroids (i,j) of these dots in the images with MATLAB. We then fed the 24 5-tuples (x,y,z,i,j) into the Tsai-Lenz camera calibration algorithm [Tsai87] which reported focal length, CCD sensor element aspect ratio, principle point, and extrinstic rotational orientation. We ignored radial lens distortion, which was reported as less than 1 pixel per 1000 pixels. Finally, we resampled the raw 256 654x496 images down to 327x248 before using them as input to the renderer.

Internally, our light fields used no compression techniques as presented in earlier papers. Our particular high frequency filter looked for the mean energy in a sampled radiance function that had been processed by a gradient magnitude Sobel edge detection filter [Lim90]. The scores on the focal surface were rendered at 920x690 and not interpolated. We use bilinear interpolation between samples on the image surfaces. When rendering with the variable apertures, we used cone weighting to combine the rays from each camera.

Using our renderer, we can typically render images with the four nearest cameras in about one second. In our code, we have maintained a general interface that allows for flexibilty and fast development. In the near future we plan to implement a real time renderer that will optimize for speed.

Conclusions

Previous implementations have tried to solve focusing problems by 1) using scenes that were roughly planar, 2) using aperture filtering to blur the input data, or 3) using approximate geometry. Unfortunately, most scenes can not be confined to a single plane, aperture filtering can not be undone or controlled at run time, and proxy surfaces can be hard to obtain. We have presented a new parameterization that allows run-time control of what should be in focus. In addition to describing focus control through aperture size and a moving focal surface, we have presented a strategy for using multiple focal planes and methods to create them. This new parameterization allows light fields to capture data sets with depth, and helps bring us closer to truly photorealistic virtual reality.

There is much future work to be done. We would like to improve our scoring system for our multiple focal planes: we need a method to differentiate between high frequencies in the radiance function caused by occlusion and those caused by empty space. In Figure 20, the errors surrounding the red bull identify how these errors affect the final images. Also, we would like an algorithm for optimally picking the *n* best focal planes, perhaps using the presented auto-focus techniques. The camera calibration step is somewhat tedious, and we would like to self-calibrate using the light fields. This would give us the optimal camera model for each light field, as opposed to assuming the light field to works with a prior camera calibration. Finally, we are working on methods to speed up the renderer so that we can view these light fields in a head-mount-display.

We would like to thank Hughes Research Labs, Intel Corporation, and Microsoft Corporation for monetary, equipment, and software donations. Also, thanks to Neil Alexander for his *"Alexander Bay"* tree model, and to Charles Lee for help in making our pictures and animations.

Figure 17: Using a single focal plane, only the red bull is in focus, while the yellow fish and the monitor are out of focus.

Figure 18: By moving the focal plane back in the scene, we can make the fish in focus, while the red bull and the monitor are out of focus.

Figure 19: Finally, the focal plane is at the back of the room, making the monitor in focus, while the fish and red bull are blurry.

Figure 20: By using two focal planes, we can make the bull and the monitor in focus, while the regions in between are still out of focus.

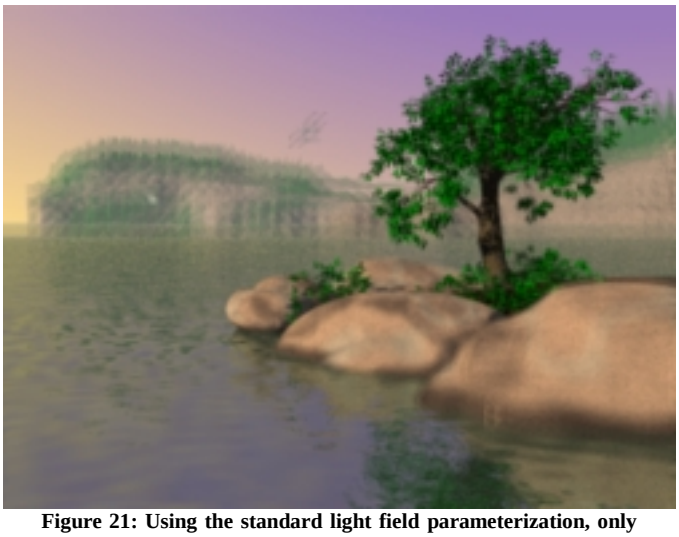


Figure 21: Using the standard light field parameterization, only one fixed plane can be in focus. Using the smallest aperture available, this would be the best picture we could create.

Figure 22: By using 4 focal planes (originally 8 focal planes with scores compressed down to 4), we can clearly do better than the image in Figure 21. Especially note the hills in the background and the rock in the foreground.

References

[Bolles87] Bolles, R. C., H. H. Baker, and D. H. Marimont, "*Epipolar-Plane Image Analysis: An Approach to Determining Structure from Motion*," **International Journal of Computer Vision**, Vol. 1, 1987.

[Camahort98] Camahort, E., A. Lerios, and D. Fussell, "*Uniformly Sampled Light Fields*," **Proceedings of the 9th EUROGRAPHICS Workshop on Rendering**, Vienna, Austria, June/July 1998

[Cook84] Cook, R.L., T. Porter, and L. Carpenter, "*Distributed Ray Tracing*," **Computer Graphics** (SIGGRAPH'84 Conference Proceedings), July 1984, pp. 137-145.

[Gortler96] Gortler, S.J., R. Grzeszczuk, R. Szeliski, and M.F. Cohen, "*The Lumigraph*," **Computer Graphics** (SIGGRAPH'96 Conference Proceedings), August 1996, pp. 43-54.

[Gu96] Gu, X., S.J. Gortler, M.F. Cohen, "*Polyhedral Geometry and the Two-Plane Parameterization*," **7th Eurographics Workshop on Rendering**, 1996. (also, http://hillbilly.deas.harvard.edu/~sjg/papers/tpp.ps)

[Haeberli94] Haeberli, Paul, "*A Multifocus Method for Controlling Depth of Field*," http://www.sgi.com/grafica/depth/index.html, October 1994.

[Levoy96] Levoy, M. and P. Hanrahan, "*Light Field Rendering*," **Computer Graphics** (SIGGRAPH'96 Conference Proceedings), August 1996, pp. 31-42.

[Lim90] Lim, J.S., **Two-dimensional Signal and Image Processing**, Prentice Hall P T R, New Jersey, 1990, pp 476 –483.

[Mitchell88] Mitchell, D.P. and A.N. Netravali, "*Reconstruction Filters in Computer Graphics*," **Computer Graphics** (SIGGRAPH '88 Conference Proceedings), August 1988, pp. 221-228

[Pentland87] Pentland, A.P., "*A New Sense for Depth of Field*," **IEEE Transactions on Pattern Analysis and Machine Intelligence**, vol. 9, no. 4, July 1987, pp. 523-531.

[Tsai87] Tsai, R. Y., "*A Versatile Camera Calibration Technique for High-Accuracy 3D Machine Vision Metrology Using Off-the-Shelf TV Cameras and Lenses*," **IEEE Journal of Robotics and Automation**, Vol. RA-3, No. 4, August 1987.

Appendix E:

Polyhedral Visual Hulls for Real-Time Rendering

Wojciech Matusik Chris Buehler Leonard McMillan
MIT Laboratory for Computer Science

Abstract. We present new algorithms for creating and rendering visual hulls in real-time. Unlike voxel or sampled approaches, we compute an exact polyhedral representation for the visual hull directly from the silhouettes. This representation has a number of advantages: 1) it is a view-independent representation, 2) it is well-suited to rendering with graphics hardware, and 3) it can be computed very quickly. We render these visual hulls with a view-dependent texturing strategy, which takes into account visibility information that is computed during the creation of the visual hull. We demonstrate these algorithms in a system that asynchronously renders dynamically created visual hulls in real-time. Our system outperforms similar systems of comparable computational power.

1 Introduction

A classical approach for determining a three-dimensional model from a set of images is to compute shape-from-silhouettes. Most often, shape-from-silhouette methods employ discrete volumetric representations [12, 19]. The use of this discrete volumetric representation invariably introduces quantization and aliasing artifacts into the resulting model (i.e. the resulting model seldom projects back to the original silhouettes).

Recently, algorithms have been developed for sampling and texturing visual hulls along a discrete set of viewing rays [10]. These algorithms have been developed in the context of a real-time system for acquiring and rendering dynamic geometry. These techniques do not suffer from aliasing effects when the viewing rays correspond to the pixels in a desired output image. In addition, the algorithms address the rendering problem by view-dependently texturing the visual hull with proper visibility. However, these algorithms are only useful when a view-dependent representation of the visual hull is desired.

In this paper, we present algorithms for computing and rendering an exact polyhedral representation of the visual hull. This representation has a number of desirable properties. First, it is a view-independent representation, which implies that it only needs to be computed once for a given set of input silhouettes. Second, the representation is well-suited to rendering with graphics hardware, which is optimized for triangular mesh processing. Third, this representation can be computed and rendered just as quickly as sampled representations, and thus it is useful for real-time applications.

We demonstrate our visual hull construction and rendering algorithms in a real-time system. The system receives input from multiple video cameras and constructs visual hull meshes as quickly as possible. A separate rendering process asynchronously renders these meshes using a novel view-dependent texturing strategy with visibility.

1.1 Previous Work

Laurentini [8] introduced the *visual hull* concept to describe the maximal volume that reproduces the silhouettes of an object. Strictly, the visual hull is the maximal volume constructed from all possible silhouettes. In this paper (and in almost any practical setting) we compute the visual hull of an object with respect to a finite number of silhouettes. The silhouette seen by a pinhole camera determines a three-dimensional volume that originates from the camera's center of projection and extends infinitely while passing through the silhouette's contour on the image plane. We call this volume a silhouette cone. All silhouette cones exhibit the hull property in that they contain the actual geometry that produced the silhouette. For our purposes, a visual hull is defined as the three-dimensional intersection of silhouette cones from a set of pinhole silhouette images.

Visual hulls are most often computed using a discrete three-dimensional grid of volume elements (voxels). This technique, known as voxel carving [12, 19], proceeds by projecting each voxel onto each of the source image planes, and removing those voxels that fall completely outside of any silhouette. Octree-hierarchies are often used to accelerate this procedure. Related to voxel approaches, a recent algorithm computes discrete slices of the visual hull using graphics hardware for acceleration [9]. Other approaches improve the shape using splines [17] or color information [18].

If the primary purpose of a shape representation is to produce new renderings of that shape from different viewing conditions, then construction of an explicit model is not necessary. The image-based visual hull technique introduced in [10], renders unique views of the visual hull directly from the silhouette images, without constructing an intermediate volumetric or polyhedral model. This is accomplished by merging the cone intersection calculation with the rendering process, resulting in an algorithm similar in spirit to CSG ray casting [15].

However, sometimes an explicit three-dimensional model of the visual hull is desired. There has been work [4, 14] on general Boolean operations on 3D polyhedra. Most of these algorithms require decomposing the input polyhedra into convex polyhedra. Then, the operations are carried out on the convex polyhedra. By contrast, our algorithm makes no convexity assumptions; instead we exploit the fact that each of the intersection primitives (i.e., silhouette cones) are generalized cones with constant scaled cross-section. The algorithm in [16] also exploits the same property of silhouette cones, but exhibits performance unsuitable for real-time use.

View-dependent rendering is very popular for models that are acquired from real images (e.g., see [13]). The rendering algorithm that we use is closely related to view-dependent texture mapping (VDTM), introduced in [5] and implemented in real-time in [6]. The particular algorithm that we use is different from those two, and it is based on the unstructured lumigraph rendering (ULR) algorithm in [3]. In our implementation, we extend the ULR algorithm to handle visibility, which was not covered in the original paper.

Our real-time system is similar to previous systems. The system in [11] constructs visual hull models using voxels and uses view-dependent texture mapping for rendering, but the processing is done as an off-line process. The Virtualized Reality system [7] also constructs models of dynamic event using a variety of techniques including multi-baseline stereo.

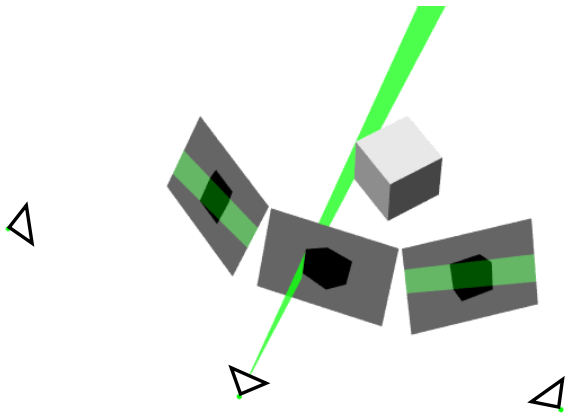


Fig. 1. A single silhouette cone face is shown, defined by the edge in the center silhouette. Its projection in two other silhouettes is also shown.

2 Polyhedral Visual Hull Construction

We assume that each silhouette s is specified by a set of convex or non-convex 2D polygons. These polygons can have holes. Each polygon consists of a set of edges joining consecutive vertices that define its (possibly multiple) contours. Moreover, for each silhouette s we know the projection matrix associated with the imaging device (e.g., video camera) that generated the silhouette. We use a 4×4 projection matrix that maps 3D coordinates to image (silhouette) coordinates, and whose inverse maps image coordinates to 3D directions.

2.1 Algorithm Outline

In order to compute the visual hull with respect to the input silhouettes, we need to compute the intersection of the cones defined by the input silhouettes. The resulting polyhedron is described by all of its faces. Note that the faces of this polyhedron can only lie on the faces of the original cones, and the faces of the original cones are defined by the projection matrices and the edges in the input silhouettes.

Thus, a simple algorithm for computing the visual hull might do the following: For each input silhouette s_i and for each edge e in the input silhouette s_i we compute the face of the cone. Then we intersect this face with the cones of all other input silhouettes. The result of these intersections is a set of polygons that define the surface of the visual hull.

2.2 Reduction to 2D

The intersection of a face of a cone with other cones is a 3D operation (these are polygon-polyhedron intersections). It was observed by [10, 16] that these intersections can be reduced to simpler intersections in 2D. This is because each of the silhouette cones has a fixed scaled cross-section; that is, it is defined by a 2D silhouette. Reduction to 2D also allows for less complex 2D data structures to accelerate the intersections.

To compute the intersection of a face f of a cone $cone(s_i)$ with a cone $cone(s_j)$, we project f onto the image plane of silhouette s_j (see Figure 1). Then we compute the intersection of projected face f with silhouette s_j. Finally, we project back the resulting polygons onto the plane of face f.

2.3 Efficient Intersection of Projected Cones and Silhouettes

In the previous section, we discussed intersecting a projected cone face f with a silhouette s_j. If we repeat this operation for every projected cone face in $cone(s_i)$, then we will have intersected the entire projected silhouette cone $cone(s_i)$ with silhouette s_j. In this section we show how to efficiently compute the intersection of the projected cone $cone(s_i)$ with the silhouette s_j. We accelerate the intersection process by pre-processing the silhouettes into Edge-Bin data structures as described in [10]. The Edge-Bin structure spatially partitions a silhouette so that we can quickly compute the set of edges that a projected cone face intersects. In the following, we abbreviate $cone(s_i)$ as c_i for simplicity.

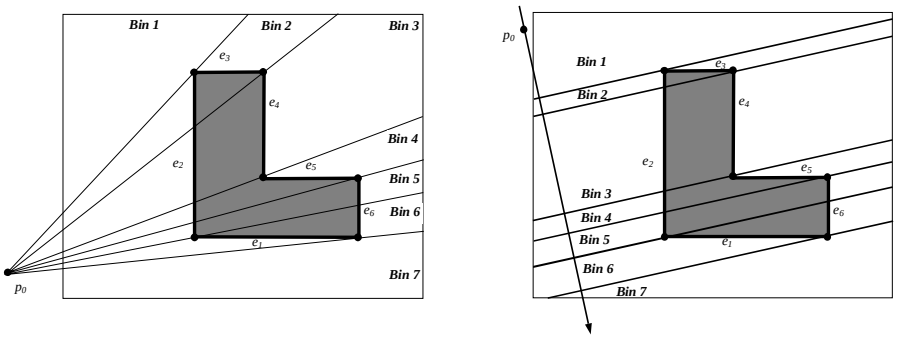
Construction of Edge-Bins. First, we observe that in case of perspective projection all rays on the surface of the cone c_i project to a pencil of lines sharing a common point p_0 (i.e., the epipole) in the image plane of s_j. We can parameterize all projected lines based on the slope α that these lines make with some reference line. Given this parameterization we partition the domain of $\alpha = (-\infty, \infty)$ into ranges such that any projected line with the slope falling inside of the given range always intersects the same set of edges of the silhouette s_j. We define a bin b_k to be a three-tuple: the start α_{start}, the end α_{end} of the range, and a corresponding set of edges S_k, $b_k = (\alpha_{start}, \alpha_{end}, S_k)$. We note that each silhouette vertex corresponds to a line that defines a range boundary.

In certain configurations, all rays project to a set of parallel epipolar lines in the image plane of s_j. When this case occurs, we use a line $p(\alpha) = p_0 + d\alpha$ to parameterize the lines, where p_0 is some arbitrary point on the line $p(\alpha)$ and d is a vector perpendicular to the direction of the projected rays. To define bins, we use the values of the parameter α at the intersection points of the line $p(\alpha)$ with the epipolar lines passing through the silhouette vertices. In this way we can describe the boundary of the bin using two values α_{start} and α_{end}, where α_{start}, α_{end} are the values of α for the lines passing through two silhouette vertices that define the region.

The Edge-Bin construction involves two steps. First, we sort the silhouette vertices based on the value of the parameter α. The lines that pass through the silhouette vertices define the bin boundaries.

Next, we observe that two consecutive slopes in the sorted list define α_{start} and α_{end} for each bin. To compute a set of edges assigned to each bin we traverse the sorted list of silhouette vertices. At the same time we maintain the list of edges in the current bin. When we visit a vertex of the silhouette we remove from the current bin an edge that ends at this vertex, and we add an edge that starts at the vertex. The start of an edge is defined as the edge endpoint that has a smaller value of parameter α. In Figure 2 we show a simple silhouette, bins, and corresponding edges for each bin.

The edges in each bin need to be sorted based on the increasing distance from the point p_0 (or the distance from parameterization line $p(\alpha)$ in case of the parallel lines). The efficient algorithm first performs a partial ordering on all the edges in the silhouette such that the edges closer to the point p_0 are first in the list. Then, when the bins are constructed the edges are inserted in the bins in the correct order.



Bin	1	2	3	4	5	6	7
Edges	$\emptyset$	e_2, e_3	e_2, e_4	e_2, e_5	e_2, e_6	e_1, e_6	$\emptyset$

Fig. 2. Two example silhouettes and their corresponding Edge-Bin data structures. Two cases are shown, one with convergent bins and one with parallel bins. The edges that are stored in the bins are listed in the accompanying table.

Efficient Intersection of the Projected Cone Faces with a Silhouette. Using the edge bin data structure, we can compute efficiently the intersection of the projected cone c_i with the silhouette s_j of some other cone c_j. In order to compute the intersection we process the faces of cone c_i in consecutive order. We start by projecting the first face f_1 onto the plane of silhouette s_j. The projected face f_1 is defined by its boundary lines with the values α_{p1} , α_{p2}. First, we need to find a bin $b = \{\alpha_{start}, \alpha_{end}, S\}$ such that $\alpha_{p1} \in (\alpha_{start}, \alpha_{end})$. Then, we intersect the line α_{p1} with all the edges in S. Since the edges in S are sorted based on the increasing distance from the projected vertex of cone c_i (or distance from line $p(\alpha)$ in case of parallel lines) we can immediately compute the edges of the resulting intersection that lie on line α_{p1}. Next, we traverse the bins in the direction of the value α_{p2}. As we move across the bins we build the intersection polygons by adding the vertices that define the bins. When we get to the bin $b' = \{\alpha'_{start}, \alpha'_{end}, S'\}$ such that $\alpha_{p2} \in (\alpha'_{start}, \alpha'_{end})$ we intersect the line α_{p2} with all edges in S' and compute the remaining edges of the resulting polygons. It is important to note that the next projected face f_2 is defined by the boundary lines α_{p2}, α_{p3}. Therefore, we do not have to search for the bin α_{p2} falls into. In this manner we compute the intersection of all projected faces of cone c_i with the silhouette s_j.

2.4 Calculating Visual Hull Faces

In the previous section we described how to perform the intersection of two cones efficiently. Performing the pairwise intersection on all pairs of cones results in $k - 1$ polygon sets for each face of each cone, where k is the total number of silhouettes. The faces of the visual hull are the intersections of these polygon sets at each cone face. It is possible to perform the intersection of these polygon sets using standard algorithms for Boolean operations [1, 2], but we use a custom algorithm instead that is easy to implement and can output triangles directly.

Our polygon intersection routine works by decomposing arbitrary polygons into quadrilaterals and intersecting those. In Figure 3, we demonstrate the procedure with

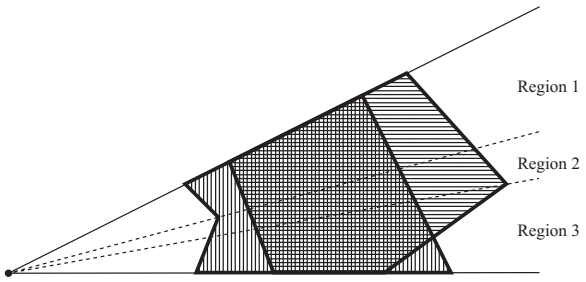


Fig. 3. Our polygon intersection routine subdivides polygons into quadrilaterals for intersection.

two 5-sided polygons, one with vertical hatching and the other with horizontal hatching. We first divide the space occupied by the polygons into triangular regions based on the polygons' vertices and the apex of the silhouette cone (similar to the Edge-Bin construction process). Note that within each triangular region, the polygon pieces are quadrilaterals. Then, we intersect the quadrilaterals in each region and combine all of the results into the final polygon, shown with both horizontal and vertical hatching.

The resulting polyhedral visual hull includes redundant copies of each vertex in the polyhedron (in fact, the number of copies of each vertex is equal to the degree of the vertex divided by 2). To optionally eliminate the redundant copies, we simply merge identical vertices. Ideally, our algorithm produces a watertight triangular mesh. However, because of our non-optimal face intersection routine, our meshes may contain T-junctions which violate the watertight property.

2.5 Visibility

In order to properly texture map the visual hull we need to determine which parts of the visual hull surface are visible from which cameras.

This visibility problem is equivalent to the shadow determination problem where one places a point light source at each reference camera position, and the goal is to determine which parts of the scene are illuminated by the light and which parts lie in a shadow. Standard graphics (hardware) algorithms are directly applicable since we have a mesh representation of the visual hull surface. However, they require rendering the scene from each input camera viewpoint and reading the z-buffer or the frame-buffer. These operations can be slow (reading the frame and z-buffer can be slow) and they can suffer from the quantization artifacts of the z-buffer.

We present an alternative novel software algorithm that computes the visible parts of the visual hull surface from each of the input cameras. The algorithm has the advantages that it is simple, and it can be computed virtually at no cost at the same time that we compute the visual hull polygons.

Let us assume that we want to compute whether the faces of the visual hull that lie on the extruded edge i in silhouette s_j are visible from image k.

We observe that these faces have to be visible from the camera k if the edge i is visible from the epipole p_0 (the projection of the center of projection of image k onto the image plane of camera j). This effectively reduces the 3D visibility computation to the 2D visibility computation. Moreover, we can perform the 2D visibility computation very efficiently using the edge-bin data structures that we already computed during the

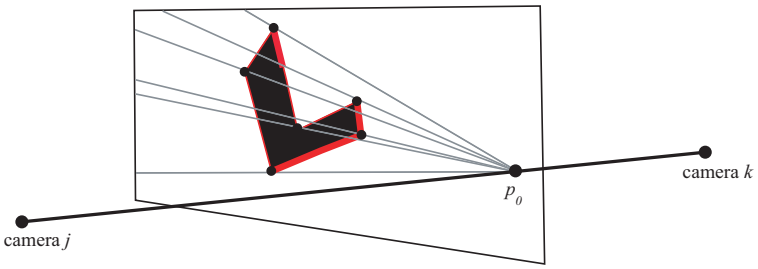


Fig. 4. We perform a conservative visibility test in 2D. In this example, the thick edges in the silhouette of camera c_j have been determined to be visible by camera c_k. These 2D edges correspond to 3D faces in the polyhedral visual hull.

visual hull computation.

First, we label all edges invisible. Then, to determine the visibility of edges in image j with respect to image k we traverse each bin in the Edge-Bin data structure. For each bin, we label the part of the first edge that lies in the bin as visible (see Figure 4). The edges in the bin are sorted in the increasing distance from the epipole; thus, the first edge in the bin corresponds to the front-most surface.

If the edge is visible in its full extent (if it is visible in all the bins in which it resides) then the edge is visible. If the edge is visible in some of its extent (if it is visible only in some bins in which it resides) then the edge is partially visible. The easiest solution in this case is to break it into the visible and invisible segments when computing the faces of the visual hull.

The visibility computation described in this section is conservative; we never label an edge visible if it is in fact invisible. However, it is often over-conservative, especially for objects whose silhouettes contains many holes.

3 View-Dependent Texturing

We have applied a novel view-dependent texturing strategy for rendering our polyhedral visual hull models in real-time. Our algorithm is based on the unstructured lumigraph rendering (ULR) algorithm detailed in [3], and we have added extensions to handle the visibility information computed during the visual hull construction.

The core idea of ULR is that the influence of a single image on the final rendering is a smoothly varying function across the desired image plane (or, equivalently, across the geometry representing the scene). These smooth weighting functions combine to form a image "blending field" that specifies how much contribution each input image makes to each pixel in the output image. The assumption of smoothness suggests an efficient rendering strategy: sparsely sample the image blending field and reconstruct it using simple basis functions (e.g., linear hat functions). The reconstructed blending field is then used to blend pixels from the input images to form the output image.

In the case of real-time rendering, the blending field can be efficiently reconstructed by triangulating the samples and using hardware alpha interpolation across the faces of

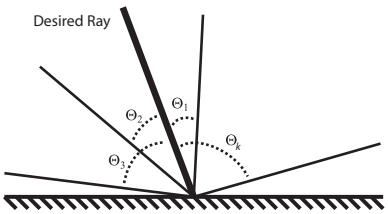


Fig. 5. Our k-nearest neighbor weighting is based on the k cameras with viewing rays closest in angle to the desired viewing ray. In this example, the desired ray is shown in bold in addition to four camera viewing rays. The angles of the k closest cameras are ordered such that $\Theta_1 \leq \Theta_2 \leq \Theta_3 \leq \ldots \leq \Theta_k$, and Θ_k is taken to be the threshold at which the weighting function falls to zero.

the triangles. The input image pixels are then blended together by projectively texturing mapping the triangles and accumulating the results in the frame buffer. The pseudocode for a multi-pass rendering version of the algorithm proceeds as follows:

```
Construct a list of blending field sample locations
for each input image i do
   for each blending field sample location do
      evaluate blending weight for image i and store in alpha channel
   end for
   Set current texture i
   Set current texture matrix P_i
   Draw triangulated samples using alpha channel blending weights
end for
```

The sample locations are simply 3D rays along which the blending field is evaluated. In the case when a reasonably dense model of the scene is available, sampling along the rays emanating from the desired viewpoint and passing through the vertices of the model is generally sufficient to capture the variation in the blending field. In this case, the triangles that are drawn are the actual triangles of the scene model. By contrast, in the general unstructured lumigraph case, one may sample rays randomly, and the triangles that are drawn may only roughly approximate the true scene geometry.

The texture matrix P_i is simply the projection matrix associated with camera i. It is rescaled to return texture coordinates between 0 and 1. In our real-time system, these matrices are obtained from a camera calibration process.

3.1 Evaluating the Blending Weights

Our view-dependent texturing algorithm evaluates the image blending field at each vertex of the visual hull model. The weight assigned to each image is calculated to favor those cameras whose view directions most closely match that of the desired view. The weighting that we use is the k-nearest neighbor weighting used in [3] and summarized here. For each vertex of the model, we find the k cameras whose viewings rays to that vertex are closest in angle to the desired viewing ray (see Figure 5). Consider the k^{th} ray with the largest viewing angle, Θ_k. We use this angle to define a local weighting function that maps the other angles into the range from 0 to 1: $weight(\Theta) = 1 - \frac{\Theta}{\Theta_k}$.

Applying this function to the k angles results (in general) in $k-1$ non-zero weights. We renormalize these weights to arrive at the final blending weights. In practice, we typically use $k = 3$ in our four camera system, which results in two non-zero weights at each vertex.

Although other weighting schemes are possible, this one is easy to implement and does not require any pre-processing such as in [6]. It results in a (mostly) smooth blending field except in degenerate cases, such as when k (or more) input rays are equidistant from the desired ray or when less than k nearest neighbors can be found (due to visibility or some other reason).

3.2 Handling Visibility

The algorithm in [3] does not explicitly handle the problem of visibility. In our case, we have visibility information available on a per-polygon basis. We can distinguish two possible approaches to incorporating this information: one that maintains a continuous blending field reconstruction and one that does not. A continuous blending field reconstruction is one in which the blending weights for the cameras on one side of a triangle edge are the same as on the other side of the edge. A continuous reconstruction generally has less apparent visual artifacts.

A simple rule for utilizing visibility while enforcing continuous reconstruction is the following: if vertex v belongs to *any* triangle t that is not visible from camera c, then do not consider c when calculating the blending weights for v. This rule causes camera c's influence to be zero across the face of triangle t, which is expected because t is not visible from c. It also forces c's influence to fall to zero along the other sides of the edges of t (assuming that the mesh is watertight) which results in a continuous blending function.

The assumption of a watertight mesh makes the continuous visibility rule unsuitable for our non-watertight visual hull meshes. Even with a watertight mesh, the mesh must be fairly densely tessellated, or the visibility boundaries may not be well-represented.

For these reasons, we relax the requirement of reconstruction continuity in our visibility treatment. When computing blending weights, we create a separate set of blending weights for each triangle. Each set of blending weights is computed considering only those cameras that see the triangle. When rendering, we replicate vertices so that we can specify different sets of blending weights per-triangle rather than per-vertex. Although this rendering algorithm is less elegant and more complex than the continuous algorithm, it works well enough in practice.

4 Real-Time System

The current system uses four calibrated Sony DFW-V500 IEEE-1394 video cameras. Each camera is attached to a separate client (600 MHz Athlon desktop PC). The cameras are synchronized to each other using an external trigger signal. Each client captures the video stream at 15 fps and performs the following processing steps: First, it segments out the foreground object using background subtraction. Then, the silhouette and texture information are compressed and sent over a 100Mb/s network to a central server. The system typically processes video at 320×240 resolution. It can optionally process 640×480 video at a reduced frame rate.

The central server (2x933MHz Pentium III PC) performs the majority of the computations. The server application has the following three threads:

- *Network Thread* - receives and decompresses the textures and silhouettes from the clients.
- *Construction Thread* - computes the silhouette simplification, volume intersection, and visibility.
- *Rendering Thread* - performs the view-dependent texturing and display.

Each thread runs independently of the others. This allows us to efficiently utilize the multiple processors of the server. It also enables us to render the visual hull at a faster rate than we compute it. As a result, end users perceive a higher frame rate than that at which the model is actually updated.

5 Results

Our system computes polyhedral visual hull models at a peak 15 frames per second, which is the frame rate at which our cameras run. The rendering algorithm is decoupled from the model construction, and it can run up to 30 frames per second depending on the model complexity. The actual frame rates of both components, especially rendering, are dependent on the model complexity, which in turn depends on the complexity of the input silhouette contours. In order to maintain a relatively constant frame rate, we simplify the input silhouettes with a coarser polygonal approximation. The amount of simplification is controlled by the current performance of the system.

In Figure 6, we show two flat-shaded renderings of a polyhedral visual hull that was captured in real-time from our system. These images demonstrate the typical models that our system produces. The main sources of error in creating these models is poor image segmentation and a small number of input images.

Figure 7 shows the same model view-dependently textured with four video images. In Figure 7a, the model is textured using information from our novel visibility algorithm. This results in a discontinuous reconstruction of the blending field, but it more accurately captures regions of the model that were not seen by the video cameras. In Figure 7b, the model is textured without visibility information. The resulting blending field is very smooth, although some visibility errors are made near occlusions.

Figure 8 shows visualizations of the blending fields of the previous two figures. Each of the four cameras is assigned a color (red, green, blue, and yellow), and the colors are blended together using the camera blending weights. It is clear from these images that the image produced using visibility information is discontinuous while the other image is not.

6 Future Work and Conclusions

In offline testing, our algorithms are sufficiently fast to run at full 30 frames per second on reasonable computer hardware. The maximum frame rate of our current live system is limited by the fact that our cameras can only capture images at 15 frames per second in synchronized mode. Clearly, it would improve the system to use better and more cameras that can run at 30 frames per second. Additional cameras would both improve the shape of the visual hulls and the quality of the view-dependent texturing.

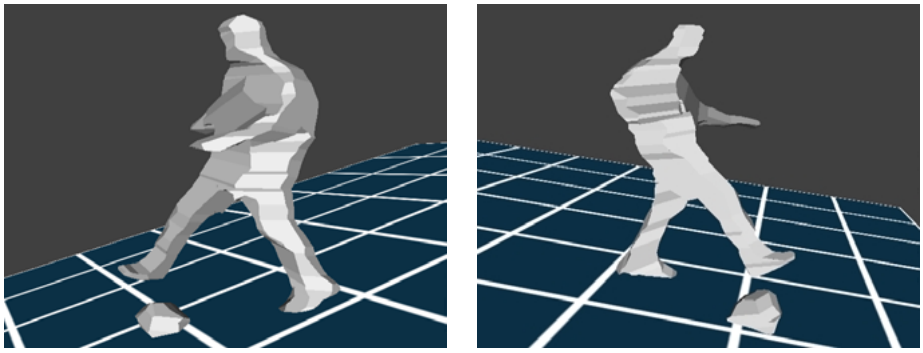
In the current system we compute and throw away a different mesh for each frame of video. For some applications it might be useful to derive the mesh of the next frame as a transformation of the mesh in the original frame and to store the original mesh plus the transformation function. Temporal processing such as this would also enable us to

accumulate the texture (radiance) of the model as it is seen from different viewpoints. Such accumulated texture information could be used to fill in parts that are invisible in one frame with information from other frames.

In this paper we have presented novel algorithms for efficiently computing and rendering polyhedral visual hulls directly from a set of images. We implemented and tested these algorithms in a real-time system. The speed of this system and the quality of the renderings are much better than previous systems using similar resources. The primary advantage of this system is that it produces polygonal meshes of the visual hull in each frame. As we demonstrated, these meshes can be rendered quickly using view-dependent texture mapping and graphics hardware.

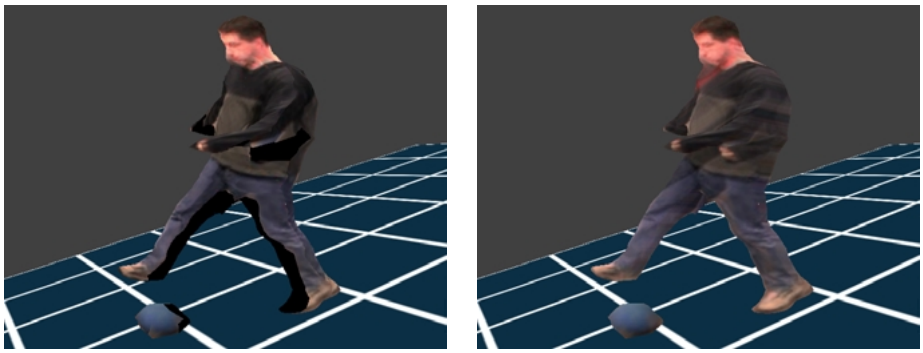
References

1. Balaban, I. J., "An Optimal Algorithm for Finding Segments Intersections," *Proc.* 11^{th} *Annual ACM Symposium on Computational Geometry*, (1995), pp. 211-219.
2. Bentley, J. and Ottmann, T., "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Trans. Comput.*, C-28, 9 (Sept. 1979), pp. 643-647.
3. Buehler, C., Bosse, M., Gortler, S., Cohen, M., McMillan, L., "Unstructured Lumigraph Rendering," To appear *SIGGRAPH 2001*.
4. Chazelle, B., "An Optimal Algorithm for Intersecting Three-Dimensional Convex Polyhedra," SIAM J. Computing, 21 (1992), pp. 671-696.
5. Debevec, P., Taylor, C., Malik. J., "Modeling and Rendering Architecture from Photographs," *SIGGRAPH 1996*, pp. 11-20.
6. Debevec, P., Yu, Y., Borshukov, G. D., "Efficient View-Dependent Image-Based Rendering with Projective Texture Mapping," *Eurographics Rendering Workshop*, (1998).
7. Kanade, T., P. W. Rander, P. J. Narayanan. "Virtualized Reality: Constructing Virtual Worlds from Real Scenes," *IEEE Multimedia*, 4, 1 (March 1997), pp. 34-47.
8. Laurentini, A., "The Visual Hull Concept for Silhouette Based Image Understanding," *IEEE PAMI*, 16, 2 (1994), pp. 150-162.
9. Lok, B., "Online Model Reconstruction for Interactive Virtual Environments," *I3D 2001*.
10. Matusik, W., Buehler, C., Raskar, R., Gortler, S., McMillan, L., "Image-Based Visual Hulls,"*SIGGRAPH 2000*, (July 2000), pp. 369-374.
11. Moezzi, S., D.Y. Kuramura, R. Jain. "Reality Modeling and Visualization from Multiple Video Sequences," *IEEE CG&A*, 16, 6 (Nov 1996), pp. 58-63.
12. Potmesil, M., "Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images," *CVGIP*, 40 (1987), pp. 1-29.
13. Pulli, K., Cohen, M., Duchamp, T., Hoppe, H., Shapiro, L., and Stuetzle, W., "View-based Rendering: Visualizing Real Objects from Scanned Range and Color Data," *8th Eurographics Workshop on Rendering*, 1997.
14. Rappoport, A. and Spitz, S., "Interactive Boolean Operations for Conceptual Design of 3D Solids," *SIGGRAPH 1997*, pp. 269-278.
15. Roth, S. D., "Ray Casting for Modeling Solids," *Computer Graphics and Image Processing*, 18 (Feb 1982), pp. 109-144.
16. Rozenoer, M. and Shlyakhter, I., "Reconstruction of 3D Tree Models from Instrumented Photographs," M.Eng. Thesis, M.I.T., (1999).
17. Sullivan, S. and Ponce, J., "Automatic Model Construction, Pose Estimation, and Object Recognition from Photographs Using Triangular Splines," *ICCV '98*, pp. 510-516, 1998.
18. Seitz, S. and Dyer, C., "Photorealistic Scene Reconstruction by Voxel Coloring," *CVPR '97*, pp. 1067-1073, 1997.
19. Szeliski, R., "Rapid Octree Construction from Image Sequences," *CVGIP: Image Understanding*, 58, 1 (July 1993), pp. 23-32.



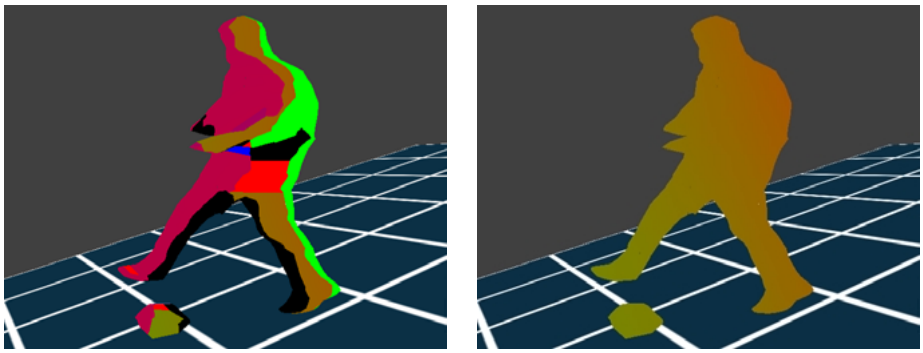
(a) (b)

Fig. 6. Two flat-shaded views of a polyhedral visual hull.



(a) (b)

Fig. 7. Two view-dependently textured views of the same visual hull model. The left rendering uses conservative visibility computed in real-time by our algorithm. The right view ignores visibility and blends the textures more smoothly but with potentially more errors.



(a) (b)

Fig. 8. Two visualizations of the camera blending field. The colors red, green, blue, and yellow correspond to the four cameras in our system. The blended colors demonstrate how each pixel is blended from each input image using both (a) visibility and (b) no visibility.

Appendix F:

Unstructured Lumigraph Rendering

Chris Buehler Michael Bosse Leonard McMillan Steven Gortler Michael Cohen
MIT Laboratory for Computer Science Harvard University Microsoft Research

Abstract

We describe an image based rendering approach that generalizes many current image based rendering algorithms, including light field rendering and view-dependent texture mapping. In particular, it allows for lumigraph-style rendering from a set of input cameras in arbitrary configurations (i.e., not restricted to a plane or to any specific manifold). In the case of regular and planar input camera positions, our algorithm reduces to a typical lumigraph approach. When presented with fewer cameras and good approximate geometry, our algorithm behaves like view-dependent texture mapping. The algorithm achieves this flexibility because it is designed to meet a set of specific goals that we describe. We demonstrate this flexibility with a variety of examples.
Keyword Image-Based Rendering

1 Introduction

Image-based rendering (IBR) has become a popular alternative to traditional three-dimensional graphics. Two effective IBR methods are view-dependent texture mapping (VDTM) [3] and the light field/lumigraph [10, 5] approaches. The light field and VDTM algorithms are in many ways quite different in their assumptions and input. Light field rendering requires a large collection of images from cameras whose centers lie on a regularly sampled two-dimensional patch, but it makes few if any assumptions about the geometry of the scene. In contrast, VDTM assumes a relatively accurate geometric model, but requires only a small number of images from input cameras that can be in general positions. These images are then "projected" onto the geometry for rendering.

We suggest that, at their core, these two approaches are quite similar. Both are methods for interpolating color values for a desired ray as some combination of input rays. In VDTM this interpolation is performed using a geometric model to determine which pixel from each input image "corresponds" to the desired ray in the output image. Of these corresponding rays, those that are closest in angle to the desired ray are weighted to make the greatest contribution to the interpolated result.

Light field rendering can be similarly interpreted. For each desired ray (s, t, u, v), one searches the image database for rays that intersect near some (u, v) point on a "focal plane" and have a similar angle to the desired ray, as measured by the ray's intersection on the "camera plane" (s, t). In a depth-corrected lumigraph, the focal plane is effectively replaced with an approximate geometric model, making this approach even more similar to view dependent texture mapping.

Given these related IBR approaches, we attempt to address the following questions: Is there a generalized rendering framework that spans all of these image-based rendering algorithms, having VDTM and lumigraph/light fields as extremes? Might such an algorithm adapt well to various numbers of input images from cameras in general configurations while also permitting various levels of geometric accuracy?

In this paper we approach the problem by suggesting a set of goals that any image based rendering algorithm should have. We find that no previous IBR algorithm simultaneously satisfies all of these goals. Therefore these algorithms behave quite well under appropriate assumptions on their input, but may produce unnecessarily poor renderings when these assumptions are violated.

We then describe an algorithm for "unstructured lumigraph rendering" (ULR), that generalizes both lumigraph and VDTM rendering. Our algorithm is designed specifically with the stated goals in mind. As a result, our renderer behaves well with a wide variety of inputs. These include source cameras that are not on a common plane, such as source images taken by moving forward into a scene, a configuration that would be problematic for previous IBR approaches.

It should be no surprise that our algorithm bears many resemblances to earlier approaches. The main contribution of our algorithm is that, unlike previously published methods, it is designed to meet a set of listed goals. Thus, it works well on a wide range of differing inputs, from few images with an accurate geometric model to many images with minimal geometric information.

2 Previous Work

The basic approach to view dependent texture mapping (VDTM) is put forth by Debevec et al. [3] in their Façade image-based modeling and rendering system. Façade is designed to estimate geometric models consistent with a small set of source images. As part of this system, a rendering algorithm was developed where pixels from all relevant cameras were combined and weighted to determine a view-dependent texture for the derived geometric models. In later work, Debevec et al [4] describe a real-time VDTM algorithm. In this algorithm, each polygon in the geometric model maintains a "view map" data structure that is used to quickly determine a set of three input cameras that should be used to texture it. Like most real-time VDTM algorithms, this algorithm uses hardware supported projective texture mapping [6] for efficiency.

At the other extreme, Levoy and Hanrahan [10] describe the light field rendering algorithm, in which a large collection of images are used to render novel views of a scene. This collection of images is captured from cameras whose centers lie on a regularly sampled two-dimensional plane. Light fields otherwise make few assumptions about the geometry of the scene. Gortler et al. [5] describe a similar rendering algorithm called the lumigraph. In addition, the authors of the lumigraph paper suggest many workarounds to overcome limitations of the basic approach, including a "rebinning" process to handle source images acquired from general camera positions and a "depth-correction" extension to allow for more ac-

curate ray reconstructions from an insufficient number of source cameras.

Many extensions, enhancements, alternatives, and variations to these basic algorithms have since been suggested. These include techniques for rendering digitized three-dimensional models in combination with acquired images such as Pulli et al. [13] and Wood et al. [18]. Shum et al. [17] suggests alternate lower dimensional lumigraph approximations that use approximate depth correction. Heigl et al. [7] describe an algorithm to perform IBR from an unstructured set of data cameras where the projections of the source cameras' centers were projected into the desired image plane, triangulated, and used to reconstruct the interior pixels. Isaksen et al. [9] show how the common "image-space" coordinate frames used in light field rendering can be viewed as a focal plane for dynamically generating alternative ray reconstructions. A formal analysis of the trade off between the number of cameras and the fidelity of geometry is presented in [1].

3 Goals

We begin by presenting a list of desirable properties that we feel an ideal image-based rendering algorithm should have. No previously published method satisfies all of these goals. In the following section we describe a new algorithm that attempts to meet these goals while maintaining interactive rendering rates.

Use of geometric proxies: When geometric knowledge is present, it should be used to assist in the reconstruction of a desired ray (see Figure 1). We refer to such approximate geometric information as a *proxy*. The combination of accurate geometric proxies with nearly Lambertian surface properties allows for high quality reconstructions from relatively few source images. The reconstruction process merely entails looking for rays from source cameras that see the "same" point. This idea is central to all VDTM algorithms. It is also the distinguishing factor in geometry-corrected lumigraphs and surface light field algorithms. Approximate proxies, such as the focal-plane abstraction used by Isaksen [9], allow for the accurate reconstruction of rays at specific depths from standard light fields.

With a highly accurate geometric model, the visibility of any surface point relative to a particular source camera can also be determined. If a camera's view of the point is occluded by some other point on the geometric model, then that camera should not be used in the reconstruction of the desired ray. When possible, image-based algorithms should consider visibility in their reconstruction.

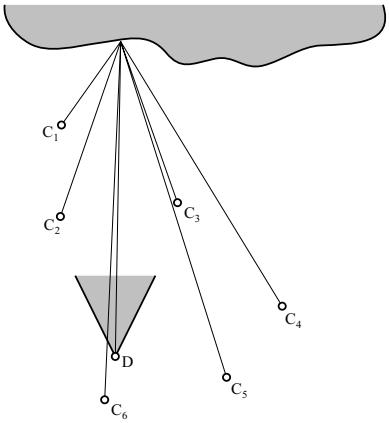


Figure 1: When available, approximate geometric information should be used to determine which source rays correspond well to a desired ray. Here C_x denotes the position of a reference camera, and D is desired novel viewpoint.

Unstructured input: It is also desirable for an image-based rendering algorithm to accept input images from cameras in general position. The original light field method assumes that the cameras are arranged at evenly spaced positions on a single plane. This limits the applicability of this method since it requires a special capture gantry that is both expensive and difficult to use in many settings [11].

The lumigraph paper describes an acquisition system that uses a hand-held video camera to acquire input images [5]. They apply a preprocessing step, called rebinning, that resamples the input images from virtual source cameras situated on a regular grid. This rebinning process adds an additional reconstruction and sampling step to lumigraph creation. This extra step tends to degrade the overall quality of the representation. This can be demonstrated by noting that a rebinned lumigraph cannot, in general, reproduce its input images. The surface light field algorithm suffers from essentially the same resampling problem.

Epipole consistency: When a desired ray passes through the center of projection of a source camera it can be trivially reconstructed from the ray database (assuming a sufficiently high-resolution input image and the ray falls within the camera's field-of-view) (see Figure 2). In this case, an ideal algorithm should return a ray from the source image. An algorithm with epipole consistency will reconstruct this ray correctly without any geometric information. With large numbers of source cameras, algorithms with epipole consistency can create accurate reconstructions with essentially no geometric information. Light field and lumigraph algorithms are designed specifically to maintain this property.

Surprisingly, many real-time VDTM algorithms do not ensure this property, even approximately, and therefore, will not work properly when given poor geometry. The algorithms described in [13, 2] reconstruct all of the rays in a fixed desired view using a fixed selection of three source images but, as shown by the original light field paper, proper reconstruction of a desired image may involve using some rays from each of the source images. The algorithm described in [4] always uses three source cameras to reconstruct all of the desired pixels on a polygon of the geometry proxy. This departs from epipole consistency if the proxy is coarse. The algorithm of Heigl et al. [7] is an notable exception that, like a light field or lumigraph, maintains epipole consistency.

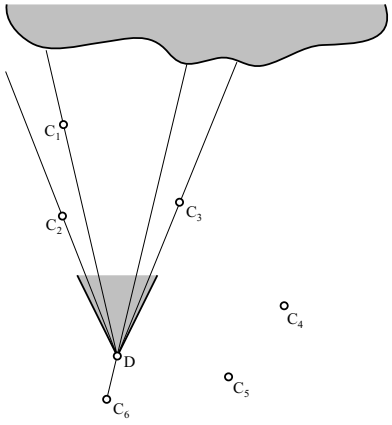


Figure 2: When a desired ray passes through a source camera center, that source camera should be emphasized most in the reconstruction. Here this case occurs for cameras C_1, C_2, C_3, and C_6.

Minimal angular deviation: In general, the choice of which input images are used to reconstruct a desired ray should be based on a natural and consistent measure of closeness (See Figure 3). In particular, source image rays with similar angles to the desired ray should be used when possible.

Interestingly, the light field and lumigraph rendering algorithms that select rays based on how close the ray passes to a source camera manifold do not quite agree with this measure. As shown in figure 3, the "closest" ray on the (s,t) plane is not necessarily the closest one measured in angle.

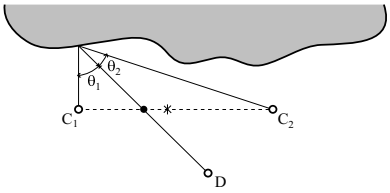


Figure 3: Angle deviation is a natural measure of ray difference. Interestingly, as shown in this case, the two plane parameterization gives a different ordering of "closeness." Source camera C_2's ray is closer in angle to the desired ray, but the ray intersects the camera (s,t) plane closer to C_1.

Continuity: When one requests a ray with infinitesimal small distance from a previous ray intersecting a nearby point on the proxy, the reconstructed ray should have a color value that is correspondingly close to the previously reconstructed color. Reconstruction continuity is important to avoid both temporal and spatial artifacts. For example, the contribution due to any particular camera should fall to zero as one approaches the boundary of its field-of-view [3], or as one approaches a part of a surface that is not seen by a camera due to visibility [14].

The VDTM algorithm of [4], which uses a triangulation of the directions to source cameras to pick the "closest three" does not provide spatial continuity, even at high tessellation rates of the proxy. Nearby points on the proxy can have very different triangulations of the "source camera view map" resulting in very different reconstructions. While this objective is subtle, it is nonetheless important, since lack of such continuity can introduce noticeable artifacts.

Resolution sensitivity: In reality, image pixels are not really measures of a single ray, but instead an integral over a set of rays subtending a small solid angle. This angular extent should ideally be accounted for by the rendering algorithm (See Figure 4). For example, if a source camera is far away from an observed surface, then its pixels represent integrals over large regions of the surface. If these ray samples are used to reconstruct a ray from a closer viewpoint, an overly blurred reconstruction will result (assuming the desired and reference rays subtend comparable solid angles). Resolution sensitivity is an important consideration when combining source rays from cameras with different focal lengths, or when combining rays from cameras with varying distance and obliqueness relative to the imaged surface. It is seldom considered in traditional light field and lumigraph rendering, since the source cameras usually have common focal lengths and are located roughly the same distance from any reconstructed surface. However, when using unstructured input cameras, a wider variation in camera-to-surface distances can arise, and it is important to consider image resolution in the ray reconstruction process. To date, no image-based rendering approaches have dealt with this problem.

Equivalent ray consistency: Through any empty region of space, the ray along a given line-of-sight should be reconstructed consistently, regardless of the viewpoint position (unless dictated by other goals such as resolution sensitivity or visibility) (See Figure 5). This is not the case for unstructured rendering algorithms that use desired-image-space measurements of "ray closeness" [7]. As shown in Figure 5, two desired cameras that share a desired ray will have a different "closest" cameras, therefore giving different reconstructions.

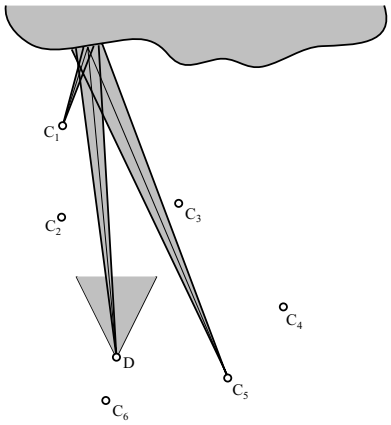


Figure 4: When cameras have different views of the proxy, their resolution differs. Here cameras C_1 and C_5 see the same proxy point with different resolutions.

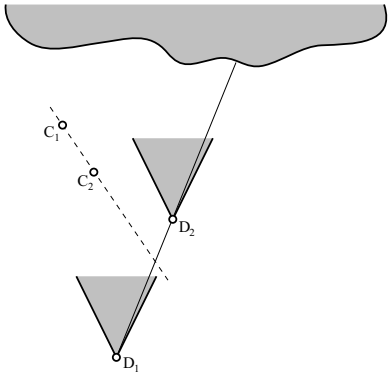


Figure 5: When ray angle is measured in the desired view, one can get different reconstructions for the same ray. The algorithm of Heigl et al. would determine C_2 to be the closest camera for D_1, and C_1 to be the closest camera for D_2. The switch in reconstructions occurs when the desired camera passes the dotted line.

Real-time: It is desirable that the rendering algorithm run at interactive rates. Most of the image-based algorithms that we considered here achieve this goal. In designing a new algorithm to meet our desired goals we have also strived to ensure that the result is still computed efficiently.

Table 1 summarizes the goals of what we would consider an ideal rendering method. It also compares our Unstructured Lumigraph Rendering (ULR) method to other published methods.

4 Unstructured Lumigraph Rendering

We present a lumigraph rendering technique that directly renders views from an unstructured collection of input images. The input to our Unstructured Lumigraph Rendering (ULR) algorithm is a collection of source images along with their associated camera pose estimates as well as an approximate geometric proxy for the scene.

4.1 Camera Blending Field

Our real-time rendering algorithm works by first evaluating a "camera blending field" at a set of vertices in the desired image plane and interpolating this field over the whole image. This blending field describes how each source camera is weighted to reconstruct a given pixel. The calculation of this field is based on our stated

Goals	lh96	gor96	deb96	pul97	deb98	pigh98	hei99	wood00	ULR
Use of Geometric Proxy	n	y	y	y	y	y	y	y	y
Epipole Consistency	y	y	y	n	n	n	y	y	y
Resolution Sensitivity	n	n	n	n	n	n	n	n	y
Unstructured Input	n	resamp	y	y	y	y	y	resamp	y
Equivalent Ray Consistency	y	y	y	y	y	y	n	y	y
Continuity	y	y	y	y	n	y	y	y	y
Minimal Angular Deviation	n	n	y	n	y	y	n	y	y
Real-Time	y	y	n	y	y	y	y	y	y

Table 1: Comparison of the algorithms lh96 [10], gor96 [5], deb96 [3], pul97 [13], deb98 [4], pigh98 [12], hei99 [7], wood00 [18], and ULR according to our desired goals.

goals, and includes factors related to the angular difference between the desired ray and those available in the given image set, estimates of undersampling, and field-of-view [13, 12]. Given the blending field, each pixel of the desired image is then reconstructed by a weighted average of the corresponding pixels in each weighted input image.

We begin by discussing how cameras are weighted based on angle similarity. Then, we generalize our approach for other considerations such as resolution and field-of-view.

A given desired ray r_d, intersects the surface proxy at some front-most point p. We consider the rays r_i from p to the centers C_i of each source camera i. For each source camera we define the angular penalty, $\text{penalty}_{ang}(i)$, as the angular difference between r_i and r_d. (see Figure 6).

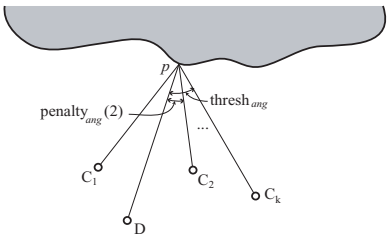


Figure 6: The angle of the k^{th} farthest camera is used as an angle threshold.

When $\text{penalty}_{ang}(i)$ is zero we would like the blending weight used for camera i, $w_{ang}(i)$, to be at a maximum. To best satisfy epipole consistency, this maximum weight should be infinite relative to the weights of all other cameras. It is unclear, however, when $w_{ang}(i)$ for a particular camera should drop to zero.

For example, one way to define a smooth blending weight would be to set a global threshold thresh_{ang}. Then, the weight $w_{ang}(i)$ could decrease from w_{max} to zero as $\text{penalty}_{ang}(i)$ increases from zero to thresh_{ang}. This approach proves unsatisfactory when using unstructured input data. In order to account for desired pixels where there are no angularly close cameras, we would need to set a large thresh_{ang}. But using a large thresh_{ang}would blend too many cameras at pixels where there are many angularly close cameras, giving an unnecessarily blurred result.

One way to solve this of problem is to use a k-nearest neighbor interpolation approach. That is, we consider only the k cameras with smallest $\text{penalty}_{ang}(\cdot)$s when reconstructing a desired ray. All other cameras are assigned a weight of zero. In this approach, we must take care that a particular camera's $w_{ang}(i)$ falls to zero as it leaves the set of closest k. We accomplish this by defining an adaptive thresh_{ang}. We define thresh_{ang}locally to be the k^{th} largest value of $\text{penalty}_{ang}(\cdot)$ in the set of k-nearest cameras. We then compute a weight function that has maximum value w_{max} at zero and has value zero at thresh_{ang}.

The blending weight that we use in our real-time system is

$$w_{ang}(i) = 1 - \frac{\text{penalty}_{ang}(i)}{\text{thresh}_{ang}}.$$

This weight function has a maximum of 1 and falls off linearly to zero at thresh_{ang}, and so consequently it does not exactly satisfy epipole consistency. Epipole consistency can be enforced by multiplying $w_{ang}(i)$ by $1/\text{penalty}_{ang}(i)$ (or by other ways) at the cost of more computation.

We then normalize the blending weights to sum to unity,

$$\tilde{w}_{ang}(i) = \frac{w_{ang}(i)}{\sum_{j=1}^{k} w_{ang}(j)}.$$

This weighting is well defined as long as all k closest cameras are not equidistant. For a given camera i, $\tilde{w}_{ang}(i)$ is a smooth function as one varies the desired ray along a continuous proxy surface.

In addition to angular difference, we also wish to penalize cameras using metrics based on resolution and field-of-view. Using these various penalties, we define the combined penalty function as

$$\begin{aligned} \text{penalty}_{comb}(i) &= \alpha\ \text{penalty}_{ang}(i) + \beta\ \text{penalty}_{res}(i) \\ &+ \gamma\ \text{penalty}_{fov}(i) \end{aligned}$$

where the constants α, β, and γ control the relative importance of the different penalties. A constant can be set to zero to ignore a penalty. We can then define $\tilde{w}_{comb}(i)$ using the k-nearest neighbor interpolation strategy described above.

Resolution Penalty Given the projection matrices of the reference cameras, the proxy point p, and the normal at p, we can predict the degree of resolution mismatch by using the Jacobian of the planar homography relating the desired view to a reference camera. This calculation subsumes most factors resulting in resolution mismatches, including distance, surface obliqueness, focal length, and output resolution.

For efficiency, we approximate this computation by considering only the distances from the input cameras to the imaged point p. In addition, we generally are only concerned with source rays r_i that significantly *undersample* the observed proxy point p. Of course, oversampling can also lead to problems (e.g., aliasing), but proper use of mip-mapping can avoid the need to penalize images for oversampling. Thus, the simplified resolution penalty function that we use is

$$\text{penalty}_{res}(i) = \max(0, \|p - C_i\| - \|p - D\|),$$

where D is the center of the desired camera.

Field-of-View Penalty We do not want to use rays that fall outside the field-of-view of a source camera. We can include this consideration using the penalty function:

$$\text{penalty}_{fov}(i) = \begin{cases} 0 & \text{if } r_i \text{ within field-of-view} \\ \infty & \text{otherwise} \end{cases},$$

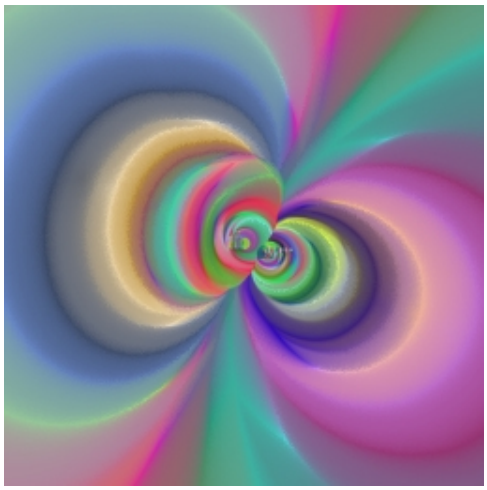


Figure 7: A visualized camera blending field. This example is from the "hallway" dataset described in the results section. The virtual camera is looking down the hallway.

which simply rejects all cameras that do not see the proxy point. In order to maintain continuity, we adjust this penalty function so that it smoothly increases toward ∞ as r_i approaches the border of image i.

With an accurate proxy, we would in fact compute visibility between p and C_i and only consider source rays that potentially see p as in [4]. In our setting we use proxies with unit depth complexity, so we have not needed to implement visibility computation. A visibility penalty function would assign ∞ to completely invisible points and small values to visible points. Care should be taken to smoothly transition from visible to invisible regions [12, 14].

In Figure 7 we visualize a camera blending field by applying this computation at each desired pixel. In this visualization, each source camera is assigned a color. The camera colors are blended at each pixel to show how they combine to define the blending field.

4.2 Real-time rendering

The basic strategy of our real-time renderer is to evaluate the camera blending field at a sparse set of points in the image plane, triangulate these points, and interpolate the camera blending field over the rest of the image (see Figure 9). This approach assumes that the camera blending field is sufficiently smooth to be accurately recovered from the samples. The pseudocode for the algorithm and descriptions of the main procedures appear below:

```
Clear frame buffer to zero
Select camera blending field sample locations
Triangulate blending field samples
for each blending field sample location j do
   for each input image i do
      Evaluate blending weight i for sample location j
   end for
   Renormalize and store k closest weights at j
end for
for each input image i do
   Set current texture to texture i
   Set current texture matrix to matrix i
   Draw triangles with blending weights in alpha channel
end for
```

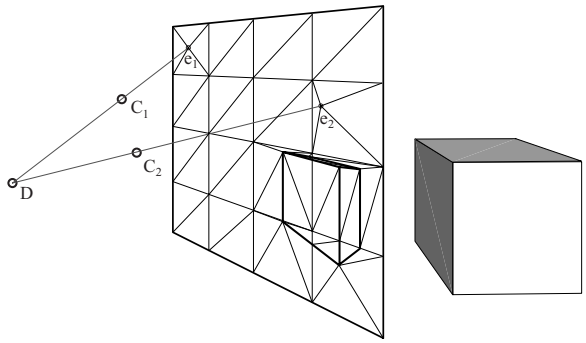


Figure 8: Our real-time renderer uses the projection of the proxy, the projection of the source camera centers, and a regular grid to triangulate the image plane.

Selecting Blending Field Samples We sample the camera blending field at a sparse set of locations in the image plane. These locations, which correspond to desired viewing rays, are chosen according to simple rules.

First, we project all of the vertices of the geometric proxy into the desired view and use these points as sample locations. To enhance epipole consistency, we next add a sample at the projection of every source camera in the desired view. Finally, we include a regular grid of samples on the desired image plane to obtain a sufficiently dense set of samples needed to capture the interesting spatial variation of the camera blending weights.

Triangulating Samples We next construct a *constrained* Delaunay triangulation of the blending field samples (see Figure 8).

First, we add the edges of the geometric proxy as constraints on the triangulation. This constraint prevents triangles from spanning two different surfaces on the proxy. Next, we add the edges of the regular grid as constraints on the triangulation. These constraints help keep the triangulation from flipping as the desired camera is moved.

Given this set of vertices and constraint edges, we create a constrained Delaunay triangulation of the image plane using Shewchuk's software [16]. The code automatically inserts new vertices at all edge-edge crossings.

Evaluating Blending Weights At each vertex of the triangulation, we compute and store the set of cameras with non-zero blending weights and their associated blending weights. Recall that at a vertex, these weights always sum to one.

Multiple sets of weights may need to be stored at each sample location if the sampling ray intersects the proxy multiple times. Triangles adjacent to these samples may need to be rendered multiple times on different proxy planes.

Drawing Triangles We render the desired image as a set of projectively mapped triangles as follows. Suppose that there are a total of m unique cameras ($k \leq m \leq 3k$, where k is the neighborhood size) with nonzero blending weights at the three vertices of a triangle.

Then this triangle is rendered m times, using the texture from each of the m cameras. When a triangle is rendered using one of the source camera's texture, each of its three vertices is assigned an alpha value equal to its weight at that vertex. The texture matrix is set to projectively texture the source camera's data onto the rendered proxy triangle. For sampling rays that intersect the proxy multiple times, the triangles associated with those samples are rendered once for each planar surface that they intersect, with the z-buffer resolving visibility.

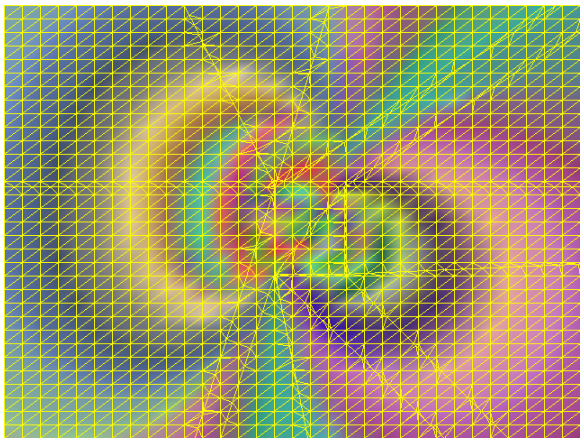


Figure 9: A visualized sampled color blending field from the real-time renderer. Camera weights are computed at each vertex of the triangulation. The sampling grid is 32×32 samples.

5 Results

We have collected a wide variety of data sets to test the ULR algorithm. In the following, we describe how the data sets are created and show some renderings from the real-time ULR algorithm. In all cases, the size k of the camera neighborhood is 4, $\alpha = 1$, $\beta = 0$, and $\gamma = 1$ unless stated otherwise. A 16×16 size grid is used for sampling the camera blending field.

Pond The pond dataset (Figure 11a) is constructed from a two second (60 frame) video sequence captured with a digital hand-held video camera. The camera is calibrated to recover the focal length and radial distortion parameters of the lens. The cameras' positions are recovered using structure-from-motion techniques.

In this simple example, we use a single plane for the geometric proxy. The position of the plane is computed based on the positions of the cameras and the positions of the three-dimensional structure points that are computed during the vision processing. Specifically, the plane is oriented (roughly) parallel to the camera image planes and placed at the average $1/z$ distance [1] from the cameras.

Since the cameras are arranged roughly along a linear path, and the proxy is a single plane, the pond dataset exhibits parallax in only one dimension. However, the effect is convincing for simulating views near the height at which the video camera was held.

Robot The Robot dataset (Figure 11b) was constructed in the same manner as the pond dataset. In fact, it is quite simple to build unstructured lumigraphs from short video sequences such as these. The robot sequence exhibits view-dependent highlights and reflections on its leg and on the tabletop.

Helicopter The Helicopter dataset (Figure 11c) uses the ULR algorithm to achieve an interesting added aspect: motion in a lumigraph. To create this "motion lumigraph", we exploit the fact that the motion in the scene is periodic.

The lumigraph is constructed from a *continuous* 30 second video sequence in which the camera is moved back and forth repeatedly over the scene. The video frames are then calibrated spatially using the structure-from-motion technique described above. The frames are also calibrated temporally by measuring the period of the helicopter. Assuming the framerate of the camera is constant, we can assign each video frame a timestamp expressed in terms of the period of the helicopter. Again, the geometric proxy is a plane.

During rendering, a separate unstructured lumigraph is constructed and rendered on-the-fly for each time instant. Since very few images occur at precisely the same phase of the period, the unstructured lumigraph is constructed over a time window. The current time-dependent rendering program (an early version of the ULR algorithm) ignores the timestamps of the images when sampling camera weights. However, it would be straightforward to blend cameras in and out temporally as the time window moves.

Knick-knacks The Knick-knacks dataset (Figure 11d) exhibits camera motion in both the vertical and horizontal directions. In this case, the camera positions are determined using a 3D digitizing arm. When the user takes a picture, the location and orientation of the camera is automatically recorded. Again the proxy is a plane, which we position interactively by "focusing" [9] on the red car in the foreground.

Car While the previous datasets primarily occupy the light field end of the image-based spectrum, the Car dataset (11e) demonstrates the VDTM aspects of our algorithm. This dataset consists of only 36 images and a 500 face polygonal geometric proxy. The images are arranged in 10 degree increments along a circle around the car. The images are from an "Exterior Surround Video" (similar to a QuicktimeVR object) database found on the carpoint.msn.com website.

The original images have no calibration information. Instead, we simply assume that the cameras are on a perfect circle looking inward. Using this assumption, we construct a rough visual hull model of the car. We simultaneously adjust the camera focal lengths to give the best reconstruction. We simplify the model to 500 faces while maintaining the hull property according to the procedure in [15]. Note that the geometric proxy is significantly larger than the actual car, and it also has noticeable polygonal silhouettes. However, when rendered using the ULR algorithm, the rough shape of the proxy is largely hidden. In particular, the silhouettes of the rendered car are determined by the images and not the proxy, resulting in a smooth contour.

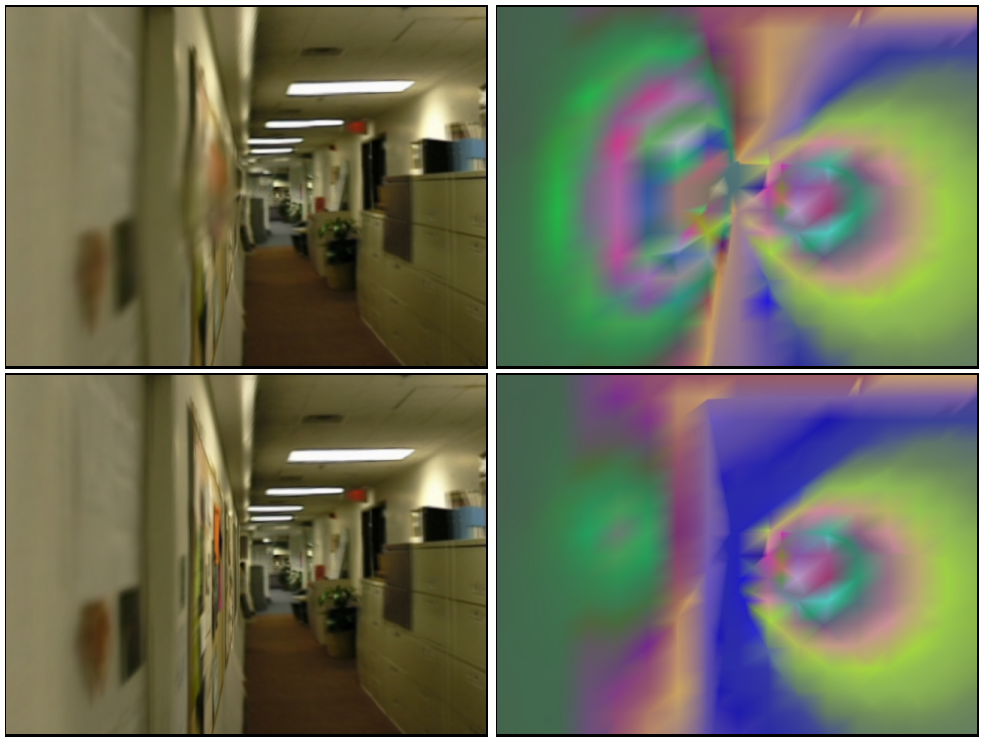
Hallway The Hallway dataset (Figure 11f) is constructed from a video sequence in which the camera moves forward into the scene. The camera is mounted on an instrumented robot that records its position as it moves. This forward camera motion is not handled well by previous image-based rendering techniques, but it is processed by the ULR algorithm with no special considerations.

The proxy for this scene is a six sided rectangular tunnel that is roughly aligned with the hallway walls [8]. None of the cabinets, doors, or other features are explicitly modeled. However, virtual navigation of the hallway gives the impression that the hallway is populated with actual three-dimensional objects.

The Hallway dataset also demonstrates the need for resolution consideration. In Figure 10a, we show the types of blurring artifacts that can occur if resolution is ignored. In Figure 10b, we show the result of using our simple resolution accommodation (β, which depends on the global scene scale, was 0.05). Low resolution images are penalized, and the wall of the hallway appears much sharper, with a possible loss of view-dependence where the proxy is poor. Below each rendering in Figure 10 appears the corresponding camera blending field. Note that 10b uses fewer images on the left hand side of the image, which is where the original rendering had most problems with excessive blurring. In this case, the removed cameras are too far behind the viewer.

6 Conclusion and Future Work

We have presented a new image-based rendering technique for rendering convincing new images from unstructured collections of input images. We have demonstrated that the algorithm can be executed efficiently in real-time. The technique generalizes lumigraph and VDTM rendering algorithms. The real-time implementation has all the benefits of structured lumigraph rendering, including



(a) (b)

Figure 10: Operation of the ULR for handling resolution issues: (a) shows the hallway scene with no consideration of resolution and (b) shows the same viewpoint rendered with consideration of resolution. Beside each image is the corresponding sampled camera blending field.

speed and photorealistic quality, while allowing for the use of geometric proxies, unstructured input cameras, and variations in resolution and field-of-view.

Many of our choices for blending functions and penalty functions are motivated by the desire for real-time rendering. More work needs to be done to determine the best possible functions for these tasks. In particular, a more sophisticated resolution penalty function is needed, as well as a more principled way to combine multiple, disparate penalties.

Further, nothing prevents our current implementation from sampling the blending field non-regularly. An interesting optimization would be to adaptively sample the blending field to better capture subtle variations and to eliminate visible grid artifacts.

Finally, not all the desired properties are created equal. It is clear that some are more important than others (e.g., equivalent ray consistency seems less important), and it would be useful to quantify these relationships for use in future algorithms.

References

[1] Jin-Xiang Chai, Xin Tong, Shing-Chow Chan, and Heung-Yeung Shum. Plenoptic sampling. *SIGGRAPH 00*, pages 307–318.

[2] Lucia Darsa, Bruno Costa Silva, and Amitabh Varshney. Navigating static environments using image-space simplification and morphing. *1997 Symposium on Interactive 3D Graphics*, pages 25–34.

[3] P. Debevec, C. Taylor, and J. Malik. Modeling and rendering architecture from photographs. *SIGGRAPH 96*, pages 11–20.

[4] Paul E. Debevec, Yizhou Yu, and George D. Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. *Eurographics Rendering Workshop 1998*.

[5] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *SIGGRAPH 96*, pages 43–54.

[6] P. Heckbert and H. Moreton. Interpolation for polygon texture mapping and shading. *State of the Art in Computer Graphics: Visualization and Modeling*, 1991.

[7] B. Heigl, R. Koch, M. Pollefeys, J. Denzler, and L. Van Gool. Plenoptic modeling and rendering from image sequences taken by hand-held camera. *Proc. DAGM 99*, pages 94–101.

[8] Y. Horry, K. Anjyo, and K. Arai. Tour into the picture: Using a spidery mesh interface to make animation from a single image. *SIGGRAPH 97*, pages 225–232.

[9] A Isaksen, L. McMillan, and S. Gortler. Dynamically reparameterized light fields. *SIGGRAPH '00*, pages 297–306.

[10] M. Levoy and P. Hanrahan. Light field rendering. *SIGGRAPH 96*, pages 31–42.

[11] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. *SIGGRAPH 2000*, pages 131–144.

[12] F. Pighin, J. Hecker, D. Lischinski, R. Szeliski, and D. H. Salesin. Synthesizing realistic facial expressions from photographs. *SIGGRAPH 98*, pages 75–84.

[13] Kari Pulli, Michael Cohen, Tom Duchamp, Hugues Hoppe, Linda Shapiro, and Werner Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. *Eurographics Rendering Workshop 1997*, pages 23–34.

[14] Ramesh Raskar, Michael S. Brown, Ruigang Yang, Wei-Chao Chen, Greg Welch, Herman Towles, Brent Seales, and Henry Fuchs. Multi-projector displays using camera-based registration. *IEEE Visualization '99*, pages 161–168.

[15] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. *SIGGRAPH 2000*, pages 327–334.

[16] Jonathan Richard Shewchuk. Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. *First Workshop on Applied Computational Geometry*, pages 124–133, 1996.

[17] Heung-Yeung Shum and Li-Wei He. Rendering with concentric mosaics. *SIGGRAPH 99*, pages 299–306.

[18] Daniel N. Wood, Daniel I. Azuma, Ken Aldinger, Brian Curless, Tom Duchamp, David H. Salesin, and Werner Stuetzle. Surface light fields for 3d photography. *SIGGRAPH 2000*, pages 287–296.

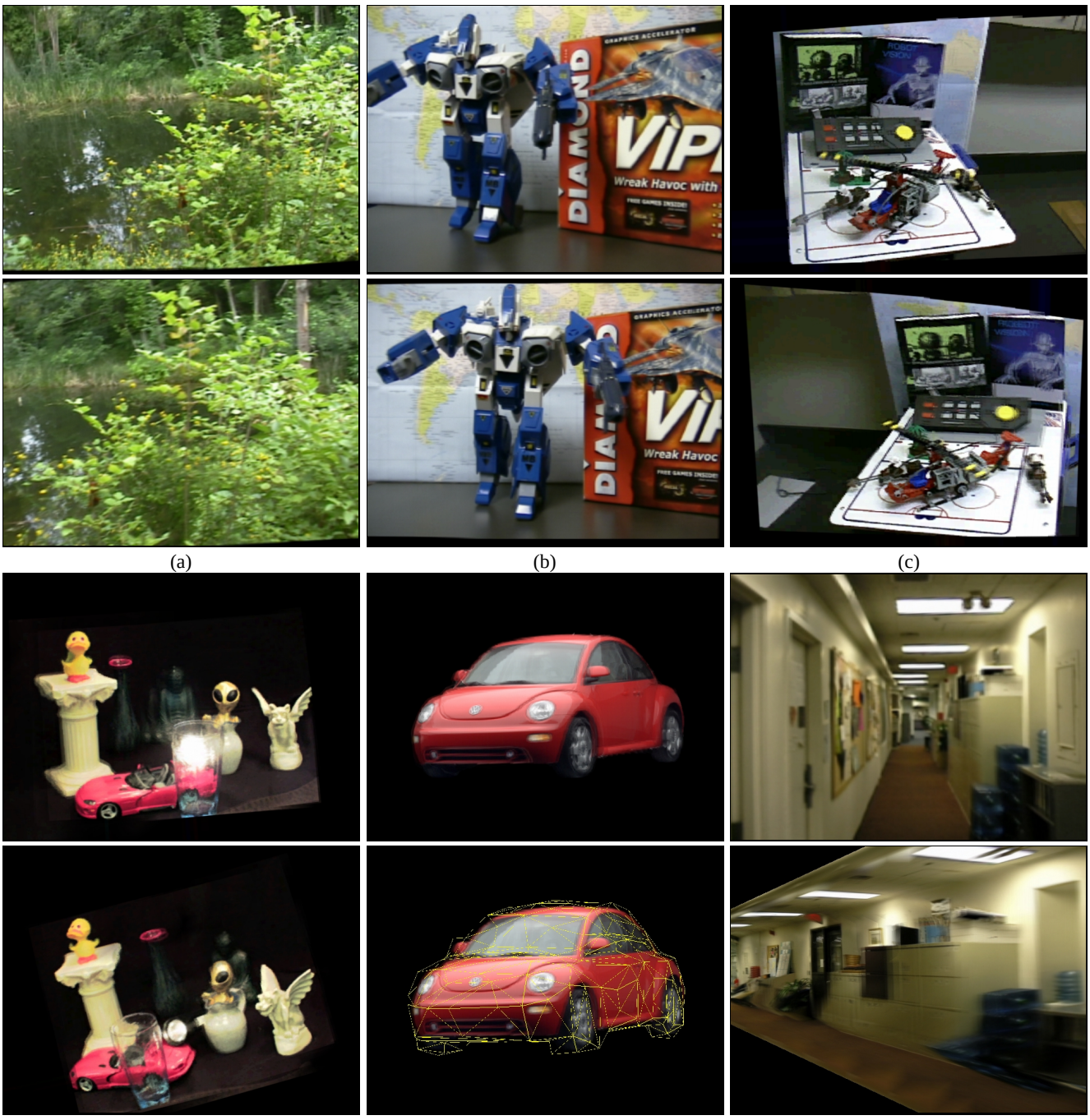


Figure 11: Renderings from the real-time unstructured lumigraph renderer. (a) and (b) show two virtual views of 60-image lumigraphs taken with a hand-held video camera. (c) shows two virtual views from a 1000-image moving lumigraph. (d) shows two virtual views of a 200-image lumigraph taken with a tracked camera. Note the active light source in the scene. (e) shows a 36-image lumigraph and its associated geometric proxy. (Original car images copyright © eVox Productions. Used with permission.) (f) shows two virtual views of a 200-image lumigraph. One virtual view is looking down the hallway, much like the input images, and one view is outside the hallway.

Appendix G:

Mesh-Based Content Routing using XML

Alex C. Snoeren, Kenneth Conley, and David K. Gifford
MIT Laboratory for Computer Science
Cambridge, MA 02139

{snoeren, conley, gifford}@lcs.mit.edu

Abstract

We have developed a new approach for reliably multicasting time-critical data to heterogeneous clients over mesh-based overlay networks. To facilitate intelligent content pruning, data streams are comprised of a sequence of XML packets and forwarded by application-level XML routers. XML routers perform content-based routing of individual XML packets to other routers or clients based upon queries that describe the information needs of downstream nodes. Our PC-based XML router prototype can route an 18 Mbit per second XML stream.

Our routers use a novel Diversity Control Protocol (DCP) for router-to-router and router-to-client communication. DCP reassembles a received stream of packets from one or more senders using the first copy of a packet to arrive from any sender. When each node is connected to n parents, the resulting network is resilient to $(n-1)$ router or independent link failures without repair. Associated mesh algorithms permit the system to recover to $(n-1)$ resilience after node and/or link failure. We have deployed a distributed network of XML routers that streams real-time air traffic control data. Experimental results show multiple senders improve reliability and latency when compared to tree-based networks.

1 Introduction

Our research is motivated by an interest in highly reliable data distribution technologies that can deliver information to end clients with low latency in the presence of both node and link failures. Low latency can be crucial for certain data that are extremely time critical. For example, real-time trading systems rely upon the timely arrival of current security prices, air-traffic control systems require up-to-the-second data on aircraft position and status, and gaps or delay in live network video and audio feeds can be distracting. In such environments, even a sub-second pause in a data feed while a delivery network retransmits or reconfigures may be unacceptable. Recent studies have shown the Internet recovers from failures on a much slower scale, often on the order of minutes [2, 20].

We observe that the achievable latency of a reliable data stream is bounded by the packet loss-recovery mechanism. Packet losses can be handled by retransmission or redundant coding. Retransmission methods limit recovery time to the round-trip delay between communicating nodes. In order to avoid retransmission in the face of loss redundant data must be sent.

This work is based upon the assumption that, in certain cases, the value of reliable and timely data delivery may justify increased transport costs if the cost increase allows us to meet a desired reliability goal. Systems often try to avoid the delay penalty by using loss-resistant coding schemes which encode redundant information into the data stream. We extend this redundancy to network delivery paths and senders. Recent work in overlay networks has shown that multiple, distinct paths often exist between hosts on the Internet [2]. We attempt to leverage these redundant network links. While some may consider this additional bandwidth wasteful, we believe the system described herein presents an interesting and elegant method of utilizing additional network resources to achieve levels of reliability and latency previously difficult to obtain.

Our basic approach is to construct a content distribution *mesh*, where every node is connected to n parents, receiving duplicate packet streams from each of its parents. The value of n is a configuration parameter that is used to select the desired trade-off between latency, reliability, and transport costs. By maintaining an acyclic mesh, this approach guarantees that the minimum cut of the mesh is n nodes or independent links. Thus, a mesh is resilient to $(n-1)$ node or $(n-1)$ independent link failures (we say $(n-1)$ resilient) without repair. If a mesh failure occurs, recovery algorithms restore the mesh to $(n-1)$ resilience in a few seconds.

Our architecture is based upon an overlay network that transports XML streams. An *XML packet* is a single independent XML document [7]. An *XML stream* is a sequence of XML packets, and each XML packet in a stream can have a different document type definition (DTD). When clients join an overlay network they specify an XML query that describes the XML packets they would like to receive. It is the job of the overlay network to configure itself to deliver the desired XML stream to a client at reasonable cost given reliability goals. Queries are expressed in a general language such as XQuery [11].

Our overlay network is implemented by XML routers. An *XML router* is a node that receives XML packets on one or more input

This research was supported in part by DARPA (Grant No. F30602-97-1-0283). Alex C. Snoeren was supported by a National Defense Science and Engineering Graduate (NDSEG) Fellowship.

links and forwards a subset of the XML packets it receives to each output link. Each output link has a query that describes the portion of the router's XML stream that should be sent to the host on that link. XML routers are components in a distributed publish-subscribe network and implement the selective forwarding of XML packets according to subscriptions described by queries.

XML has a number of advantages over a byte stream for multicast delivery. First, XML permits the network to interpret client data needs in terms of well-defined XML queries. Second, XML packets suggest what logical units of data will be processed together by a client and thus can aid network scheduling. Third, many tools and standards exist for XML making it easy for both the data originator and receiver to build robust applications. Finally, our approach allows applications and databases to push part of their processing into the network fabric. We expect that query languages such as XQuery will become standardized, allowing a single language to be used to describe data requirements. This standardization will permit applications to program our network fabric to deliver the data they need in a simple, consistent fashion.

The primary disadvantage of XML is often thought to be the increased number of bytes required to represent the same information in XML when compared to an application specific encoding. However, our experimental results suggest that conventional data compression eliminates this disadvantage. While an XML stream must be decompressed and recompressed at any router that wishes to do query matching, a router that passes all packets to every client can bypass the XML switch component entirely, and no decompression or compression need be performed. Thus, routers can include a fast-path for clients that subscribe to the unfiltered XML stream.

This paper makes three distinct, novel contributions:

- *XML Routing.* To the best of our knowledge, we describe the first packet-based network XML router to support arbitrary content routing. We believe that systems for XML routing will be useful in a wide variety of contexts and will be efficient because XML wrapper overhead can be removed by appropriate use of data compression technology.
- *Mesh-based overlay networks.* We describe the first overlay network to use multiple, redundant paths for simultaneous transport of multicast streams. Our mesh approach offers better latency performance than tree-based approaches.
- *Diversity Control Protocol.* We describe a novel protocol that uses source-independent sequence numbers to reliably reconstruct a sequenced packet stream from multiple sources. DCP reduces latency and improves reliability when compared with conventional single-sender approaches.

The remainder of this paper describes our current XML routing infrastructure in the following sections:

- Previous work (Section 2)
- Architecture of our XML routing system (Section 3)
- Mesh algorithms and distribution protocol (Section 4)
- Experimental results and our air traffic control application (Section 5)
- Issues involved in routing XML over a mesh (Section 6)
- Conclusions (Section 7)

2 Previous work

Our work on XML routers and DCP builds on a large body of past work in reliable multicast and overlay networks. We consider related work in four areas: reliable multicast, overlay networks, redundant coding and transmission schemes, and publish-subscribe networks.

2.1 Reliable multicast

Reliable multicast systems send a stream of packets to a set of receivers. Reliable multicast systems are often built on IP Multicast [3]. IP Multicast packets are duplicated by the network layer as late as possible to minimize the network resources consumed to deliver a single packet to multiple receivers. Acknowledgments are required to make IP Multicast reliable. If a packet is damaged in transmit or is lost, either a receiver will send a negative acknowledgment to the sender [14, 22, 27, 41, 43], or the lack of a positive acknowledgment from a receiver will cause the sender to retransmit [17, 22, 43]. Express [15] is a single-source multicast system that simplifies IP Multicast in the face of multiple data sources but is still integrated with the network fabric.

Of particular note is RMX [12], which shares similar goals with our work. RMX provides real-time reliable multicast to heterogeneous clients through the use of application-specific transcoding gateways. For example, it supports re-encoding images using lossy compression to service under-provisioned clients. By using self-describing XML tags, our architecture allows similar functionality to be provided in a general fashion by having clients with different resource constraints subscribe to different (likely non-disjoint) portions of the data stream.

2.2 Overlay networks

An *overlay network* is a virtual network fabric that is implemented by application level routers that communicate with each other and end clients using normal IP network facilities. Overlay networks typically use reliable point-to-point byte streams, such as TCP, to implement reliable multicast. The goal of an overlay network is typically to provide increased robustness [2, 35] or additional, sophisticated network services, such as wide-area stream broadcast [16, 30, 37], without underlying network assistance. In fact, network operators may be unaware that such services are running on their network.

One advantage of building our network as an overlay is that it is easy to modify and deploy without the cooperation of network providers. We have adopted the use of overlay networks as an effective way to build a robust mesh that can effectively route XML packets. End-system-multicast [13] is an overlay-based multicast system that constructs meshes during spanning tree discovery but does not use redundant mesh links for information delivery.

2.3 Redundant encoding and transmission

Loss-tolerant encoding schemes (often termed erasure, tornado, or forward error correcting (FEC) codes) use redundant information to support the reconstruction of a data stream in the face of a certain amount of packet loss [25]. For example, in Digital Fountain's Meta-Content protocol [9] packets are encoded to allow a receiver

to recover a data stream even if a certain fraction of Meta-Content packets are never received.

Our approach to redundancy is based on sender and channel diversity while loss-tolerant encoding schemes typically use only packet diversity [9, 33]. We use channel diversity because experimental data suggests that Internet packet errors are highly path dependent [2, 29, 35]. We use sender diversity because in single-sender systems a sender failure is likely to cause a stream gap during recovery [30]. Based on these assumptions, we believe that, with appropriately configured levels of mesh redundancy, sender and channel diversity can provide lower loss rates and latency than packet diversity, albeit at a higher cost.

Several previous systems have leveraged channel diversity, sender diversity, or both in an end-to-end fashion. Dispersity routing [24] and IDA [31] split the transfer of information over multiple network paths to provide enhanced reliability and performance. Simulation results and analytic studies have shown the benefits of this approach [5, 6]. In addition, tornado codes have been suggested to combine parallel downloads to improve reliability and performance [8]. Application-level dispersity routing, IDA, and parallel downloads use multiple network paths but do not provide for any loss recovery along a single path within the network fabric. Our use of application-level routers allows us to perform loss recovery inside of the network fabric and, thus, improve loss resilience. Further, the block encoding scheme used by Digital Fountain may add additional latency during decoding. We discuss our loss resilience results in Section 5.

2.4 Publish-subscribe systems

Publish-subscribe networks, such as Tibco's TIB^{TM}Rendezvous [28], Elvin4 [36], Siena [10], Gryphon [4], and XMLBlaster [1] permit receivers to specify the portion of a data stream that they would like to receive. Receivers typically subscribe to messages using a query that summarizes their interests. Streams may be encoded such that the same content, but in varying levels of fidelity, may be requested by each client [26, 42]. Siena and Gryphon both provide distributed implementations of singly connected graphs for information distribution, but neither provides XML-based routing.

XMLBlaster [1] is a publish-subscribe system based on XML packet streams, but it only permits queries over a specific header field. Our semantics permit queries over any field in an XML packet. We believe that the overhead of making each XML packet a fully formatted document is a small price to pay for the resulting flexibility and rational query semantics. This is especially the case when data compression causes the markup overhead in each XML packet to become negligible. To our knowledge, no existing stream-based publish-subscribe network uses redundant meshes for reliability or performance enhancement.

3 Resilient mesh networks

As shown in Figure 1, a typical overlay network for routing an XML stream contains one or more root routers (R1-R2), a variable number of internal routers (I1-I3), and a variable number of edge clients (C1-C3). Root routers are the origin of data and are assumed to have independent means of generating their XML stream. Internal routers receive the XML stream from their parent routers and forward elements of the stream to their children as required. Clients connect to routers and provide a query that describes the portions of the XML stream they would like to receive.

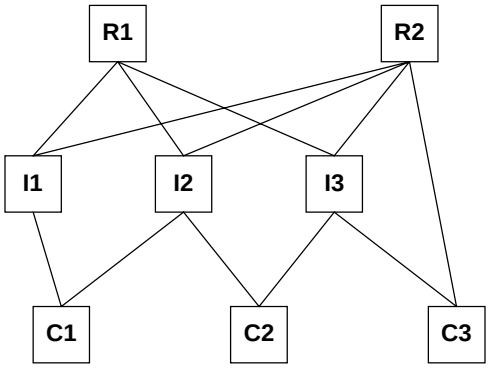


Figure 1: A mesh network comprising root routers (R1-R2), inner XML routers (I1-I3) and clients (C1-C3).

The content carried by routers in a mesh can be statically or dynamically configured. Typically, with static configuration the internal routers carry all of the XML packets available from the root routers. Thus, with a static approach to content configuration clients have a wide choice of routers that can service their request without reconfiguration delay. Unfortunately, such a mesh requires a fixed bandwidth capacity throughout. We can leverage the expressive power of XML to better control bandwidth usage.

Dynamic content configuration allows a router to carry only the packet stream necessary to service its children. In this case, a router disjoins all of the queries it receives from its children and forwards the resulting query to its parent routers. Note that since each router combines the queries of each of its children when subscribing to its parent routers, each router need only store queries for its immediate children. This results in significant bandwidth savings when clients are uninterested in the full contents of the data stream. The disadvantage of this scheme is that the mesh may not have a sufficient number of routers that currently carry the traffic needed by a node searching for an additional parent during mesh construction or repair. If a client requests information that is not available in that portion of the mesh, there will be a delay while the mesh readjusts to supply the required information although this additional startup delay is tolerable in most situations. During reconstruction, the data should be available from the current parent set. During initialization, it simply adds a slight additional startup latency.

Clients wishing to join an $(n-1)$-resilient mesh perform four distinct operations: (1) composing an XML query that describes the data desired, (2) contacting n existing routers that can service the query, (3) sending these n routers the XML query it has composed, and (4) receiving the XML stream described by the query. One particular algorithm for discovering routers is described in Section 4.

Each router includes a query table that describes the portion of the XML stream each of its children wishes to receive. Thus, each router functions as a splitter that takes a single XML stream and refines it for each child. Often a child is only interested in a subset of a stream (such as all air traffic landing in Seattle). Expressing this desire to routers saves last-mile bandwidth and end-host processing.

Our architecture also admits XML *combining routers*. A combining router merges XML feeds from different sources into a single XML feed. This can be accomplished by simply forwarding unmodified packets from both sources, or it can involve application-specific processing. For example, in our air traffic control application we are investigating merging our XML stream of air traffic data with an XML stream of runway conditions.

We will call a node k*-resilient* when any combination of k other nodes and independent links in the mesh can fail and the node will still receive its XML stream. We say a mesh is k-resilient when all of its nodes are k-resilient. The level of resilience in a network can vary according to the needs of end clients. Although we heretofore have described a uniform mesh architecture with a fixed router fan-in of n, it is entirely possible to build meshes with non-uniform fan-in. The only constraint is that in order to assure a desired level of resilience all the way to the root, the resilience of a child's parents must be equal to, or greater than the child's desired resilience. For example, one could build a core network that is 2-resilient, and certain clients could choose to be 1-resilient. The failure of a core router will most likely reduce the resilience level of many peripheral routers and clients until the mesh can reconfigure, but the mesh will continue to provide service to all clients except those clients directly connected only to the failed node. Thus, in certain circumstances, it may make sense to improve the resilience only of key portions of a network that provide service to many clients. We are investigating issues surrounding optimal mesh configuration.

4 Algorithms and protocols

An XML router implements three key algorithms and protocols:

- *XML router core*. The XML core is the engine that receives and forwards packets according to queries. Its job is to efficiently evaluate each received XML packet against all output link queries.
- *Diversity Control Protocol (DCP)*. DCP implements resilient mesh communication by allowing a receiver to reassemble a packet stream from diverse sources.
- *Mesh initialization and maintenance*. A set of algorithms automatically organizes routers and clients into a mesh and repairs the mesh when faults occur.

4.1 XML router core

Figure 2 shows the internal structure of an XML router. An XML router consists of three major components:

- An *input component* that acquires XML streams for presentation to the XML switch. The input component is responsible for maintaining DCP connections to the parents of the router and implementing the mesh initialization and reconfiguration algorithms. In addition, the input component implements data decompression. Our input component can also connect to TCP XML streams for compatibility.

  Although, in many instances, the input component will acquire a single XML stream for routing, an input component could connect to distinct meshes and merge multiple XML streams for routing. The input component is also responsible for forwarding the disjunction of its link queries to its parents.

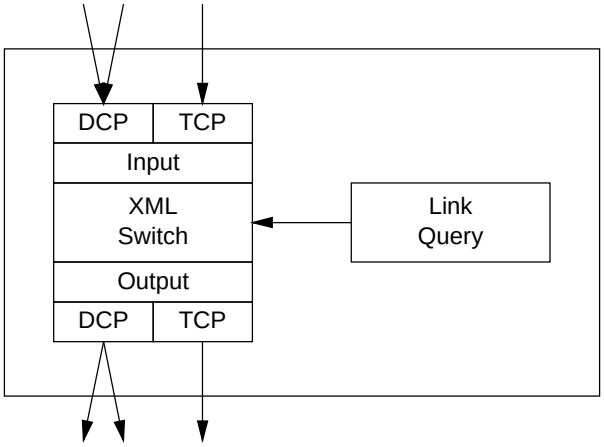


Figure 2: The internal architecture of an XML router comprises the input component, XML switch, and output component. Output link queries control XML packet forwarding.

- An *XML switch* that compares received packets against link queries, and forwards matching packets to the requesting links. An efficient XML switch attempts to combine distinct link queries into a single state machine that matches all of the link queries in a single pass over an incoming packet.
- An *output component* that forwards packets on output links using DCP. In addition, the output component is responsible for handling join requests from prospective children and implements link-based data compression. Our output component additionally can produce TCP XML streams for potential compatibility with non-DCP children.

4.2 Diversity control protocol

The *Diversity control protocol* (DCP) is so named because of the inherent sender diversity that it implements. The essential idea behind DCP is that a receiver can reassemble a packet stream from diverse senders. In DCP, the same stream of packets is sent to a receiver by multiple sources where the desired level of redundancy may vary between nodes in a mesh. As shown in Figure 3, a DCP receiver reassembles the packet stream using the first error-free packet received from any source.

4.2.1 Sequencing

Proper in-order packet stream reassembly requires that all DCP packets be assigned identifiers that admit a total ordering and that the total ordering must be known to the participants. DCP further requires identifiers obey the following invariants:

- For a given content stream, packet identifiers must be associated only with packet content and not be sender specific. This allows receivers to properly reassemble a stream based upon identifiers alone.
- Since packets may travel through a variable number of intermediate router hops, the identifiers with a particular stream must be selected at root routers and remain identifiable

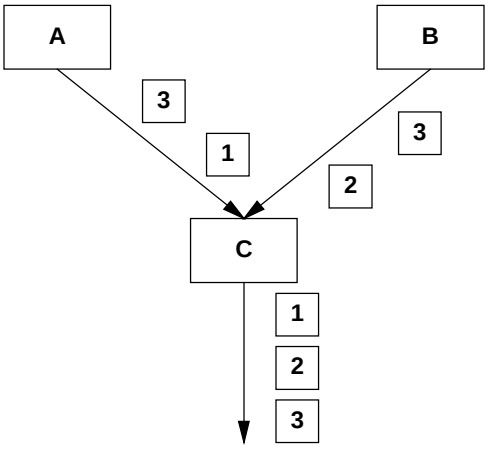


Figure 3: The Diversity Control Protocol (DCP) reassembles a packet stream from diverse senders.

throughout the mesh. Thus, the set of root routers for a particular stream must originate the same packet stream and assign the same identifiers to the same packets. This must be true even if the root routers do not generate the stream at precisely the same time or rate.

- Receiver identifier processing must admit gaps. Since intermediate routers may not forward packets containing content that was not requested by a particular receiver, the identifiers of these packets will not be received.

Our approach to satisfying these three invariants is to assign a monotonically increasing 32-bit application serial number (AN) to every DCP packet when the packet is created at a root router. Every router that forwards DCP packets maintains the last packet AN sent on each output link. The last AN sent on a link is included in the next packet along with the next packet's current AN number. Including a client-specific previous AN in each packet permits a receiver to reassemble the stream of packets from a sender in the presence of missing ANs. In our application, missing ANs are caused by filtered XML packets.

While routers may remove packets from the datagram stream, DCP itself is a reliable transport protocol. Hence, any missing datagram (as indicated by a hole in the AN sequence chain) will be retransmitted. In order to maintain redundancy invariants throughout the mesh, retransmissions are requested at each hop rather than end-to-end. Similarly, packets are buffered and transmitted in-order at each hop. This ensures that every node can consider each parent an independent source of ordered datagrams. We return to consider the implications of out-of-order forwarding in section 6.4.

DCP currently uses UDP as a transport mechanism to facilitate deployment at the application layer. Distinct DCP streams are currently transmitted on separate UDP ports. In our application, one DCP packet is used to transport one XML packet. This is possible because our XML packets are relatively small. If XML packets do not fit into a single IP packet envelope, an AN could describe both the XML packet number being transmitted and the IP packet within the XML packet. The important invariant is that an AN be based upon the content of a packet and not on when or by whom it was generated.

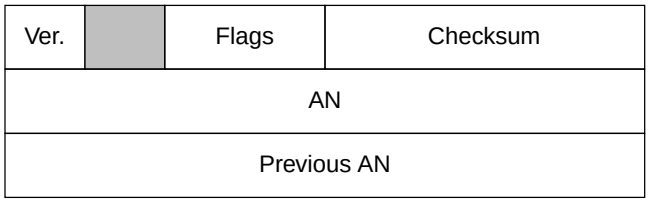


Figure 4: DCP Packet Header

Figure 4 shows the layout of a DCP packet. In addition to the ANs we have already mentioned, a DCP packet includes a 4-bit version number to allow DCP to evolve and a set of 8 bit flags. The flags permit a sender to request an acknowledgment, a receiver to send an acknowledgment or request a retransmission, and for the exchange of keep-alive and connection-establishment and tear-down information. The entire packet is covered by a 16-bit checksum which may be optionally disabled if encapsulated in UDP or when carrying streams insensitive to corruption.

While our use of DCP is as a datagram protocol, DCP is equally well-suited for the transmission of byte streams. A bit in the flags field is used to indicate that DCP is operating in stream mode. When used as a stream protocol, the AN simply refers to the sequence number of the first byte of the datagram, as in TCP. Similarly, the previous AN refers to the last byte of the previous packet in stream mode. Note that this construction allows for fragmentation or reframing of DCP packets if desired, albeit at the expense of additional complexity and buffering at the receivers. Additionally, if multiple root servers are in use each server must take care to sequence the data identically. Datagram and streaming mode cannot be used during the same DCP connection.

4.2.2 Retransmission

When a receiver joins multiple DCP senders, it waits for the first packet to arrive from any one of the hosts and uses the AN of this packet as its current AN. Packets that are subsequently received with a lower AN than the current AN are discarded and packets that are received with an AN in the future are buffered. A packet with the current AN in the previous AN field is considered the next packet in the reassembled stream and the current AN is updated. If a receiver does not receive an appropriate packet after a fixed interval, it sends a negative acknowledgment (NACK) to all senders with its current AN. This retransmission is sent only to the receiver requesting it. In a fashion similar to TCP's fast retransmit, a NACK is generated after a much shorter timeout if a packet with a subsequent AN is received, indicating either a lost or reordered packet. This NACK serves as a request for all senders to retransmit all packets after the receiver's current AN.

Assuming a regular mesh construction (equal numbers of parents and children), the negative acknowledgment process does not suffer from ACK implosion even with high degree. An individual receiver only generates a NACK if an AN is not received from any of its parents. Due to the (assumed) pairwise independence of packet loss between distinct senders and receivers, this probability drops exponentially with degree as discussed in section 5.1.2. Hence, the probability that a sender receives any NACKs at all decreases with increasing degree, avoiding the NACK implosion problem.

Senders transmit packets in order to a receiver and request an acknowledgment from a receiver from time to time. Our current implementation requests positive acknowledgment after a fixed number of packets has been sent. A receiver responds to a request for acknowledgment with an acknowledgment that contains the last AN (or last byte in streaming mode) it has processed. This serves to limit the amount of buffering required at each node and allows for rapid resynchronization of senders and receivers. If the current sender has not yet sent that AN (byte), it squelches its transmissions until after that AN (byte). A receiver can also send an unsolicited acknowledgment to squelch a sender that is behind. In contrast, if a receiver continually fails to respond to acknowledgment requests, or persistently lags behind the sequence space (indicating insufficient bandwidth between sender and receiver), the connection is terminated. The receiver must then reconnect to a new point in the mesh (presumably with a higher-bandwidth link).

4.3 Mesh formation and maintenance

A mesh begins life as a set of root routers that are all capable of supplying an XML stream of interest. We assume that failures of root routers are independent and, thus, each has an independent means of deriving the XML stream. As noted above, however, roots must be uniform in their DCP packetization and sequence number selection. Additional roots may be added to a mesh at any time provided they have a mechanism to synchronize their content stream with the existing root nodes.

Mesh discovery is outside the scope of this document, but one method of distributing the set of root nodes for a particular content stream is through DNS. All of the IP addresses for the root routers for a service could be stored in a DNS address record. For example, `stream.asdi.faa.gov` might be a DNS name that maps to a set of root routers that supply an XML stream of air traffic control data for North America.

4.3.1 Adding routers and clients

When a new internal router is added to a mesh, it can either be statically configured with a set of parents or the new router can select its own parents based upon performance experiments. A wide variety of automatic configuration algorithms can be used to form the mesh depending on the particular desires of the node. These may vary widely depending upon whether the mesh is controlled by a single administrative entity concerned with overall characteristics of the mesh such as its resilience or depth, or the new node has a more specific purpose. Clients join the mesh in the same fashion.

Rather than specify a particular algorithm or policy, we admit a host of possibilities by providing a set of mesh primitives that new nodes can use to discover the topology of the mesh and locate themselves within it. Each router supports the following primitives:

- **Join (Q):** A new node is added as a child of the router with query **Q** provided the current node is willing to admit a child with such a query.
- **Children (Q):** The router responds with its children that subscribe to a subset of **Q**. A full child list may be elicited by specifying a query that matches the entire stream.
- **Parents:** The router responds with its parent set.

1. Initialize the set S to be the root routers.
2. For each node in S, send a join request and remove the node from S.
3. If a node accepts the join, add it to the parent set P. If n nodes are in P, quit.
4. If a node declines the join, ask it for a list of its children, and add them to S.
5. If S is not empty, go to Step **2**.

Figure 5: Parent selection algorithm. Each node runs this algorithm to construct an $(n-1)$-resilient mesh.

Using these three operations, it is possible for a new node to completely walk a mesh to determine its optimal location. We note that the optimum location may vary depending on the particular desires of the joining node. We have currently implemented a very simple algorithm for automatic parent selection for a client seeking $(n-1)$ resilience shown in Figure 5.

This simple algorithm seeks to find a set of routers that are closest to the root routers and uses the timing of responses to select among candidates. The algorithm also assumes that potential parents are configured to reject join requests when they are at maximum desired capacity or they do not wish to service a requested query. We contemplate additional research on improved algorithms that are based upon both depth in the mesh and observed packet latency to select optimal parents for a new child.

Routers may refuse to serve as parents for policy reasons, if they are not receiving the portion of the XML feed necessary to service a new child's query, or if they are over-subscribed. If a prospective parent is not receiving part of the feed necessary for a new child, the prospective parent may be configured to push an expanded query up to its parent, thus propagating the information request up the mesh.

4.3.2 Mesh repair

Our mesh repair algorithm recovers from parent failures. If one of the parents of a node fails, the node actively attempts to join a new parent. The method used to obtain a new parent is currently identical to that used to obtain initial parents with one caveat. To guarantee that a mesh is acyclic, each router maintains a level number that is one greater than the maximum level of all of its parents. A router's level number is established when a router first joins the network. During mesh recovery, a router will only join parents that have a level number that is less than its own level number. If this is not possible during recovery, then a router must disconnect from all of its children and do a cold re-initialization to return to its desired level of resilience.

Our repair algorithm recovers $(n-1)$ resilience of the mesh if a non-root router fails. As discussed earlier, $(n-1)$ resilience is a fundamental property of any acyclic mesh where each child has n parents. This can be seen by forming an acyclic graph that is a dual of a mesh. In this dual graph each child is represented as a vertex that has directed edges to all of the child's parents. The min-cut of this graph is n vertices or edges if each vertex has out degree n. Thus $(n-1)$ nodes or $(n-1)$ distinct paths can fail and a node will still be connected to a root.

Due to occasional internal node failures, a mesh repaired using our algorithm will have a tendency to flatten out over time as nodes are forced to select parents with lower level numbers during each repair process. If the mesh structure is to serve extremely long-running streams, it may be necessary for nodes to occasionally remove themselves from the mesh and select an entirely new location in order to preserve the depth of the mesh and prevent overloading of root nodes. We have not yet explored efficient algorithms for determining when to start this process.

5 Evaluation

We have developed two separate implementations of our XML router. Our full-featured, multi-threaded Java implementation uses DCP for router-to-router and router-to-client communication. We have also implemented a prototype high-performance router based on Click [19]. The goal of our Java implementation was to adequately support our air traffic control application and it does not attempt to maximize absolute performance. In contrast, the Click router attempts to achieve production-grade performance using freely available XML parsing technology. Below, we report our experiences with both routers. We are mainly interested in understanding how routers will behave in a mesh under varying configurations. Thus, our evaluation focuses on the effects of mesh redundancy on DCP reliability and performance. We also provide performance results from our Click-based XML router.

5.1 DCP performance

The Diversity Control Protocol has several attractive features independent of the format of the data stream. In particular, DCP-based meshes can achieve substantially lower effective loss rates and latency than tree-based distribution networks. Further, the redundancy can be utilized to absorb unexpected decreases in the link capacity between nodes. In this section, we quantify these effects using our Java XML router. All results presented in this section represent the average of several experiments each consisting of 1000 to 10000 XML-encoded ASDI packets.

5.1.1 Experimental design

The experimental setup consisted of four 600MHz PIIIs, two running Linux 2.2.14, and two running FreeBSD 4.0. Each machine used 100Mbit Intel EtherExpress Pro100 Ethernet controllers and 128 Mbytes of memory. The roots and all intermediate XML router nodes were run on one Linux machine using Sun's JDK version 1.3. The XML client node was run on the other Linux machine, also with Sun's JDK version 1.3.

For each experiment, the root node received an XML feed containing a 1Kbyte per second substream of the live ASDI flight data described in the following section. While each node in our experimental topology requests the entire test XML stream, it does so by specifying an XML query predicate, hence each packet is parsed by the intermediate nodes as part of the forwarding process. The intermediate nodes connect to the root over the loopback interface, so there was no packet loss. The desired link loss rates between intermediate and client nodes were obtained by routing each DCP connection through one of the FreeBSD machines which passed the packets through an appropriately configured Dummynet [32] tunnel.

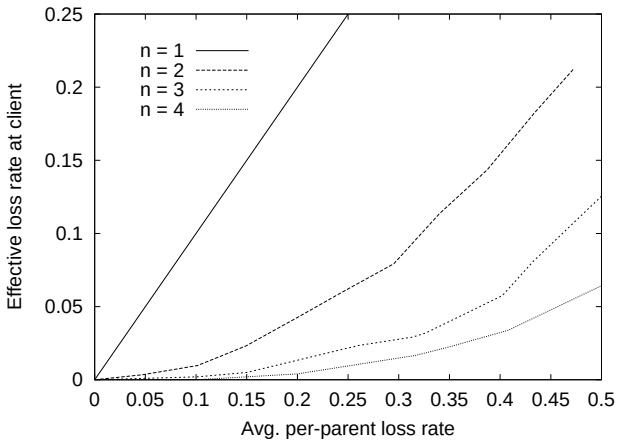


Figure 6: Loss rates experienced by a client as a function of individual parent loss rates and the number of parents.

The one-way latency between the client and parent nodes was negligible (0.1ms) with respect to the millisecond granularity of the Java-based timing mechanisms we used to measure packet latency. Variability in the observed latency of XML packets can be attributed both to the inherent non-uniformity of XML parsing times and to thread scheduling uncertainties of the JDK.

5.1.2 Redundancy reduces loss exponentially

We assume that packet loss is independent across parents. This assumption is false if a problematic portion of the communication path to a set of parents is shared. In our ideal model, if each parent has an identical loss rate, p, a node with n parents should expect a combined loss rate of p^n. Figure 6 verifies this experimentally, showing the DCP loss rate experienced at clients with 2, 3, and 4 parents where the loss rate at each parent is independently identically distributed (i.i.d.) with uniform probability p varying from $[0, 0.5]$.

For a traditional tree-based distribution network, the loss rate experienced at the client corresponds directly to the loss rate of its parent. The graph shows, however, that a mesh topology is able to provide acceptable delivery rates over even extremely lossy channels. A node with four parents can expect a loss rate of less than 5% even if each of the parents individually experiences a loss rate of up to 45%.

Most Internet links do not experience extremely high loss rates. In fact, typical long-term average loss rates are on the order of 2–5% with substantially higher burst rates [29]. In such cases, a mesh with $n = 2$ still limits the loss rate at the client to substantially less than 1%. Decreased loss rate is not the only gain from multiple parents, however.

5.1.3 Latency

Even in cases where acceptable loss rates can be provided by a tree-based network (reliability may be assured by retransmission), significant improvements in latency can be achieved by increasing redundancy. In DCP, loss is not detected until the receipt of a later packet since each individual packet is not acknowledged. However,

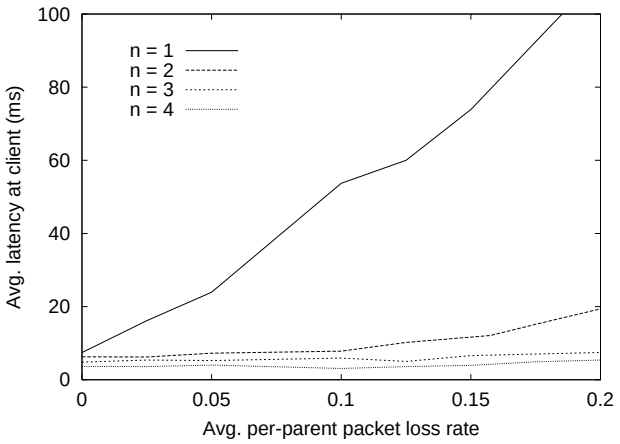


Figure 7: Average per-packet latency from root to client as a function of individual parent loss rates and the number of parents.

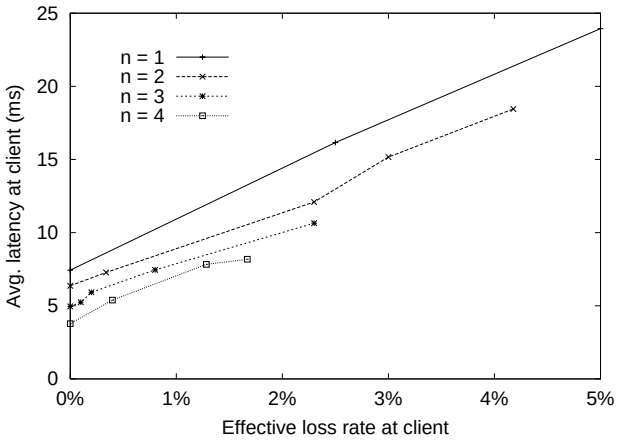


Figure 8: Average per-packet latency from root to client for a range of *effective* loss rates. Effective loss rates for each level of redundancy are taken from Figure 6.

because we expect the Internet to reorder packets [29], a packet is not assumed lost until some time after its successor arrives (currently 5ms).

In order to magnify the effects of retransmissions in a LAN environment with short round-trip times, our test XML feed was specifically constructed to have relatively long inter-packet intervals. In our experiments, a single retransmission adds approximately 300ms to the latency for the lost packet. In practice, streams are likely to have a shorter inter-arrival period but longer RTTs, resulting in a similar effect. As can be seen in Figure 7, packet loss significantly impacts the average packet latency at the client for non-redundant configurations. Meshes with higher levels of redundancy perform much better.

A redundant topology performs even better than the effective loss rate of Figure 5 suggests. This is because clients with multiple parents use the *first* copy of each XML packet they receive. In general, the expected minimum of multiple samples from any distribution

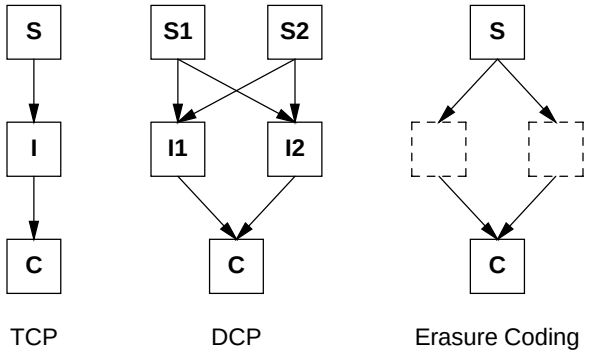


Figure 9: Experimental multi-tier topologies. A two-hop TCP path with a TCP splice in the middle, a 1-resilient DCP mesh of depth two, and an erasure code using two disjoint paths of length two with simple forwarding at the intermediate nodes. The loss rates on all links is identical.

is guaranteed to be at least as good as expected value of a single sample. Figure 8 shows the average latency for several levels of redundancy with respect to *effective* loss rates.

5.1.4 Multi-tier meshes

The improvement in latency and throughput becomes even more dramatic as the the depth of the mesh increases. We demonstrate this by measuring the throughput performance of a two-tier mesh using both DCP and TCP and analytically derive the expected performance of a carousel-based erasure code scheme. As shown in Figure 9, our experimental 1-resilient DCP mesh has five nodes: two servers delivering identical streams, two intermediate nodes, and one client. In the case of TCP, the mesh has only three nodes: a server, client, and one intermediate node that splices the two separate TCP connections. In both cases, the client, server, and intermediate nodes are connected with point-to-point Ethernet links. We analyze the performance of erasure coding over a hypothetical topology consisting of a server and client connected by two disjoint, two-hop paths. The nodes in the middle simply forward packets and, unlike TCP and DCP, do not request retransmissions of lost packets.

In our experiments we used Dummynet to limit the bandwidth of each link to 75Kbits per second and set the server to transmit data in 262-byte bursts at a rate of 19Kbits per second—significantly under link capacity. Each link in the mesh has a one-way latency of 10ms. Figure 10 shows the throughput observed at the client as the loss rate is adjusted for all links uniformly. Note that TCP's throughput drops rapidly as the loss rate increases. The redundant links are of no use to TCP as a duplicate TCP connection on a redundant path would suffer the same fate. DCP, on the other hand, is able to utilize both links at the same time to provide successful transfer at much higher loss rates.

We note that in the case of multi-hop networks with sufficient bandwidth, DCP outperforms carousel-based erasure coding techniques such as those used by Digital Fountain [9]. Such schemes do not retransmit lost packets. Instead, they encode the data stream at a fixed rate using an erasure code which enables any lost packets to be recovered by simply receiving an additional number of encoding

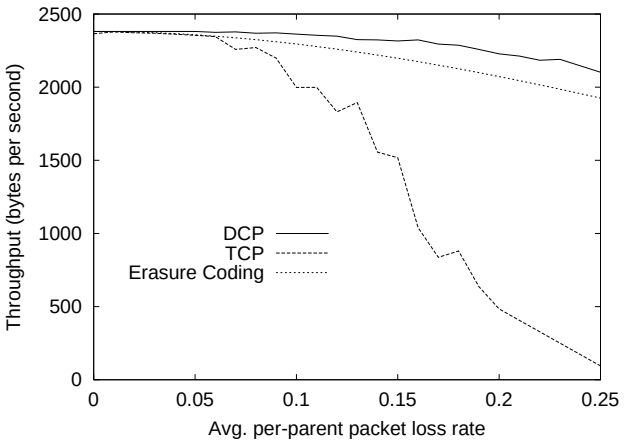


Figure 10: Observed throughput of a two-tier mesh with uniform link loss rates using both 1-resilient DCP and TCP. The stream is served in 262-byte chunks at a rate of 2381 bytes per second. DCP downloads utilize two parents at each tier while TCP can support only one at each tier. We also plot the expected performance of a simple carousel-based erasure code using two disjoint, two-hop paths.

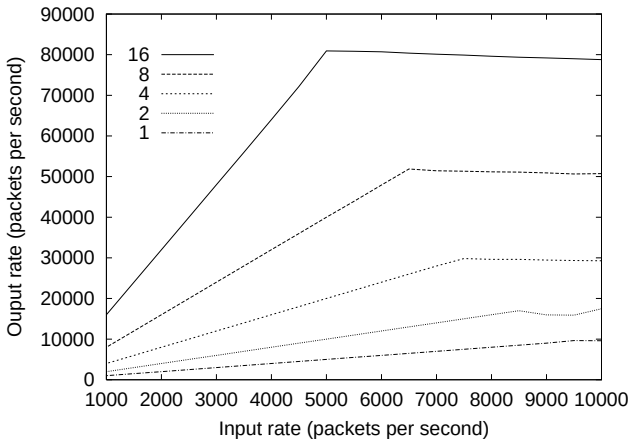


Figure 11: Forwarding rates for a Click-based XML router with a varying number of children. Each child requested the entire input XML stream specified through a trivial XPath expression.

packets. A maximum-distance-separable erasure code requires that the client simply receive as many packets as comprised the original data stream, regardless of which packets they are. In practice, many codes (including those used by Digital Fountain) are not quite maximum-distance-separable, requiring a few additional packets.

Because carousel erasure coding is typically deployed end-to-end with no retransmissions within the network, the loss rates at each hop are cumulative. Whereas, DCP reassembles the stream and retransmits a full set of redundant packets at each tier of the mesh. A naive carousel-based distribution network could support applications such as ours by ensuring each packet contains all the data necessary to decode the current input packet plus any additional redundant data required to support the erasure code.

A carousel erasure code can utilize redundant links by sending an encoded version of each data packet down all available links. Hence, we can calculate an upper bound on the performance of such an erasure code by assuming only one packet of any encoding set must be received. Given a distribution network with n separate paths, each comprised of l hops with link loss rate p, it is easy to see that each path successfully delivers the packet with probability $(1-p)^l$. In the best case, each data packet can be successfully decoded by the client if only one of the encoding packets is received, which occurs with probability $1-(1-(1-p)^l)^n$. To recover lost packets, the client must receive additional encoding packets which are lost with the same probability. Hence, the throughput of such a scheme can be computed by simply multiplying the input data rate by the effective reception rate. Using this formula, Figure 10 shows the expected performance of a sufficiently low-rate maximum-distance-separable erasure code over the two-tier topology shown in Figure 9.

5.2 XML routing performance

Figure 11 shows the forwarding rates achieved by our Click XML router installed as a kernel module. We measured the forwarding capacity by generating a constant stream of identical UDP packets containing a 262-byte XML-encoded ASDI flight update (similar to the one in Figure 12) at a fixed rate and sending them to our XML router. The router parses each XML packet, applies the appropriate child predicates, and forwards the packet to the children with matching predicates.

The tests were conducted on an 800Mhz dual-processor Intel PIII in uni-processor mode with two Ethernet controllers: an on-board Intel EtherExpress PRO 100Mbit/s PCI controller and an Intel PRO/1000 Gigabit Ethernet PCI card. We use uni-processor mode because our XML parsing code is not known to be SMP-safe. Packets were received on the 100Mbit interface and forwarded out the Gigabit interface. Because Click does not support polling on the 100Mbit controller packet input was interrupt driven.

The maximum loss-free forwarding rate varies with the number of children. Additional children add processing overhead for additional link queries. With only one client and a simple query expression, our implementation is able to forward slightly more than 9,000 262-byte packets per second, or about 19 Mbit/second. The more complicated the packet and expression, the slower the forwarding rate.

In order to better understand the impact of query complexity, we timed the XML parser and query evaluator separately. Our Click XML router uses the Gnome XML library, *libxml*. The library provides both an XML parser and an XPath (a subset of XQuery) evaluator. We have made no attempts to optimize the performance of this library. Thus, our measured performance represents a lower bound, and we expect an efficient implementation could perform much better. Table 1 shows the time taken to apply a variety of queries to the same 262-byte sample XML flight packet. We find that complex expressions can take over twice the time to evaluate in the context of an XML packet than simple ones. In all cases, packet processing cost is dominated by XML parsing time.

XQuery	Time (μs)
Parse	64.2
true()	4.5
/flight/flightleg/altitude > 300	7.1
starts-with(string(/flight/id),'TWA')	8.9
substring-before(string(/flight/flightleg \ /coordinate/lat),'N') > 2327	14.5

Table 1: Time to evaluate various queries in an 800Mhz PIII. Parsing a standard 262-byte XML flight update requires 64.2μs. The four XQuery expressions shown here select the entire feed, flights above 30,000 feet, Trans World Airlines flights, and flights currently north of the Tropic of Cancer, respectively.

5.3 Experience with air traffic control data

Our original motivation for developing XML routers was to build an infrastructure for distributing and processing real-time air traffic control data. Our laboratory receives the Aircraft Situational Display to Industry (ASDI) [40] feed via a private IP intranet connection to the U.S. Department of Transportation (DOT). The ASDI feed provides detailed information about the state of North American airspace. ASDI messages include information on flight plans, departures, flight location, and landings. A position update is received approximately once a minute for all enroute aircraft. The ASDI feed is directly distributed to most major airlines and is used for collaborative planning between the FAA and the airlines.

The ASDI feed as distributed by the DOT is encoded in ASCII with a specific compact character encoding for each ASDI message type. Efforts were made to make native ASDI messages a compressed format by virtue of their terseness. The ASDI feed is the union of feeds from multiple Air Route Traffic Control Centers (ARTCCs) and countries (USA & Canada). Unfortunately, messages that cannot be parsed using the ASDI specification arise. Thus, at the outset of our work, we built an ASDI feed parser and carefully gathered examples of non-standard messages. We slowly tuned our ASDI feed parser to handle undocumented cases and, today, still find the occasional new message format.

Early in our work, we decided to convert each ASDI message into a corresponding XML packet to create an XML packet stream. This decision guaranteed that all of our applications would have an easy to parse and well-defined XML DTD to consume. Furthermore, it centralized our interpretation of the ASDI feed so that it could be updated as new undocumented message types were identified. We call the XML stream that is created from the ASDI feed the XML ATC stream. Figure 12 shows a sample flight in both ASDI encoding and our XML encoding.

The XML-encoded ASDI packets contain widely varying amounts of data depending on the type of event being reported: flight departures, arrivals, position updates, or other auditing information. Packet size ranges from around 250 bytes to almost 1000 bytes, for an average of about 350 bytes per packet. The stream is diurnal, peaking in the early evening with an average packet inter-arrival time of about 14ms, resulting in an XML data stream of about 25 Kbytes per second.

ASDI Format:

```
153014022245CCZVTZ UAL1021 512 290 4928N/12003W
```

XML Format:

```
<?xml version="1.0"?>
<messageid>153014022245CCZVTZ</messageid>
<flight>
  <id>UAL1021</id>
  <flightleg status="active">
    <speed type="ground">512</speed>
    <altitude type="reported" mode="plain">
      290
    </altitude>
    <coordinate>
      <lat>4928N</lat>
      <lon>12003W</lon>
    </coordinate>
  </flightleg>
</flight>
```

Figure 12: The same flight data formatted in ASDI and XML. In practice, we omit the Message ID field from the XML encoding.

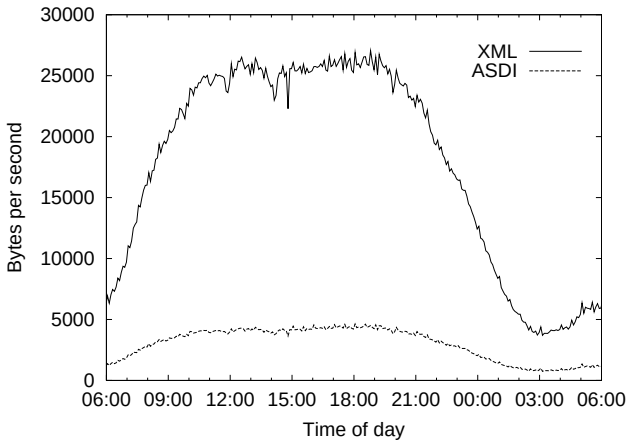


Figure 13: Average bandwidth utilization of the full XML stream and the native ASDI format vs. time of day. In both cases the Message ID field (see Figure 12) is removed from all packets at the root nodes.

Our primary concern in converting the ASDI feed to XML was potential bandwidth bloat. Figure 13 shows the bandwidth of the ASDI feed in both native and XML formats averaged over five minute intervals. Simply converting the feed to XML results in approximately a four-fold increase in bandwidth when compared to the native ASDI feed. We ran both the XML and native ASDI streams through a Lempel-Ziv [21] data compressor. Figure 14 shows the bandwidth of compressed forms of the same streams shown in Figure 13. While the ASDI feed compresses over a factor of two, the XML feed compresses over a factor of 10. The net result is a compressed XML ATC stream is only slightly larger than a compressed ASDI feed and more efficient than the raw ASDI feed.

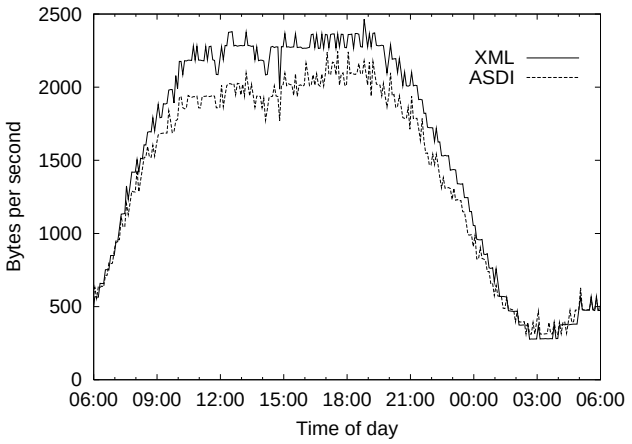


Figure 14: Average bandwidth utilization of the compressed XML stream and the compressed ASDI format vs. time of day. Again, in both cases the Message ID field is removed from all packets at the root nodes.

We have run mesh networks with two root routers and four internal routers but a single root router is more typical. This is because our DOT link is a single point of failure and terminates at our root router(s). We are adding a second communication line to the DOT to connect to their backup ASDI system. This will enable us to have two root routers with independent failure modes. Application serial numbers (ANs) in our ATC application are provided by the FAA. Hence, synchronizing multiple roots is straightforward. Each ASDI message includes a Message ID that we use as the AN of the corresponding XML packet.

Figure 15 shows one interface to the XML ATC stream. This graphical client implements DCP and connects to our XML router mesh. The panel on the left of the screen can be used to control the display of aircraft information. Different colors are used to depict aircraft altitude and the client will coast the position of an aircraft between position updates. For our particular application domain of air traffic control data, XML proved to be a robust and efficient mechanism for distribution. We anticipate adding new types of clients, including an XML stream recorder, to our current system.

6 Discussion

This section considers the strengths and weaknesses of our approach to content routing using XML. While we believe that many of the techniques we developed for our ATC application are widely applicable, we would like to make our assumptions clear.

6.1 AN generation

One difficulty in providing redundant packet sources is providing a standardized sequence space for packet streams that obey the three invariants we outlined in Section 4.2.1. Often, application-specific solutions will present themselves, such as source-derived sequence numbers or time codes. However, in the absence of application-provided sequence numbers, it is necessary to use other approaches, such as cumulative byte counts, block fingerprint matching [23, 38, 39], or other derived metrics.

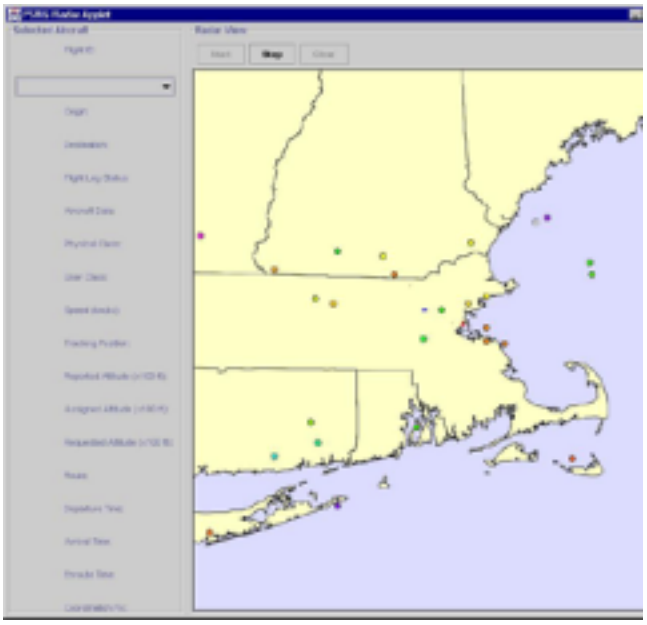


Figure 15: Java-based client for XML ATC stream showing controls and air traffic.

When a combining router merges XML streams, packets in the combined stream must have appropriate ANs. Simply using the ANs from the original uncombined packets for packets in the combined stream will typically not work as packets from different streams will in general have incomparable ANs. One solution is for the combining router to become the root of a new stream and establish its own totally ordered AN space. However, this would create a single point of failure if sequence assignments in this space are not coordinated with another combining router. Another approach is to make AN space partially ordered. For example, the AN space for a combined stream could be a pair of the AN of the source packet in its original stream along with an integer suffix that identifies the source XML stream. Packets with ANs from different source streams would not be sequenced across streams, but a client could recover the ordering of XML packets within each stream.

6.2 Flow control

Nodes are responsible for monitoring the loss rate of streams from their parent and adjusting their predicates appropriately. Limited per-child buffering is available at each node, and clients may be disconnected if they are consequently unable to consume the data stream at an acceptable rate.

The squelching mechanism of DCP allows parents to avoid wasting bandwidth sending packets to a child that the child has already received. If a child is unable to keep up with the long-term average rate of the stream, however, queues will build up and action must be taken. If the client is able to subsist with a smaller subset of the data stream, it may wish to conduct join experiments in order to determine the appropriate XML query for its bandwidth constraints [26, 42]. Otherwise, clients persistently unable to keep up with the data stream will be disconnected by their parents.

6.3 Redundancy

We expect that most mesh networks will use $n = 2$. This level of redundancy allows for single points of failure and allows mesh repair to proceed without stream flow interruption. We expect that as future networks increase in capacity a moderate amount of packet redundancy will be acceptable for high-value streams to achieve specific reliability and performance goals. Secondary storage is often replicated for similar reasons.

We have assumed in our analysis that errors from different parents are independent. This assumption can be violated in numerous ways, but the most likely reason will be shared communication path components from a child to its parents. In addition, network-wide effects, such as distributed-denial-of-service attacks, could cause independent parents to have dependent packet losses. To maximize link independence, we plan to explore using routers in distinct Internet autonomous systems (ASs) and ensuring that last-mile bandwidth is adequate to each AS. In certain applications, it may also be possible to use private intranets to better control error assumptions.

6.4 Router XML stream reassembly

Each of our routers recreates the original XML stream before it is processed by the XML switch. We do this to guarantee that every XML packet is forwarded by every router, to allow a client to ask for retransmissions from any of its parents, and to potentially allow the XML switch to keep stream-dependent state between packets that could be used by queries. The amount of buffering required is bounded by requiring positive acknowledgments as discussed in section 4.2.2.

If XML packets are forwarded out-of-order by an XML switch then a router does not necessarily need to buffer packets or recreate the original sequenced XML stream. This is, indeed, the case in our ATC application, although in our ATC application every XML router does recreate the original XML stream. If an XML router need not recreate the original XML stream, a router could process each received packet independently and would not need to process every packet in an XML stream. In this scenario, a client places increasing reliance upon the redundancy of the mesh to ensure timely delivery of packets that are not received from a particular router. In particular, since all levels may forward out of sequence, the latency induced by a retransmission request from the client may be large. Hence, we have not yet considered how to handle reliable, out-of-order delivery with bounded latency.

6.5 Packet acknowledgments

For asynchronous, variable-bandwidth data streams, packet loss can be detected either by the lack of packet acknowledgments at a sender or by a gap in packet sequence at a receiver. DCP currently relies upon the latter technique. If inter-arrival times are large, per-packet ACKs may be required to provide the appropriate level of responsiveness. Unfortunately, positive acknowledgment schemes admit a well-known implosion problem where the sender is flooded with acknowledgments from each of its children.

While our implementation currently uses unicast UDP to transport DCP packets, DCP could employ IP Multicast where available. The negative acknowledgment system we describe is capable of handling IP Multicast packet losses. If IP Multicast were employed, a DCP output component would send a single packet to an appropriate multicast group of its children based upon the children's queries.

6.6 Dynamic timer adjustment

A robust DCP implementation should be able to automatically adjust its timers to the characteristics of the link between nodes. In particular, the negative acknowledgment timer should be set only long enough to admit observed packet reordering, which clearly depends on the inter-packet arrival of the flow. Being too slow results in poor latency, being too jumpy results in wasted bandwidth. Similarly, several timers relating to mesh liveliness would benefit from automatic refinement. In particular, nodes expect to receive data from their parents every so often. If no data is available in that interval, the parent sends a keep-alive message. A similar mechanism is employed by the parent to insure the continued presence of its children. Clearly the timer should be proportional to the stream data rate, in order to avoid excessive probing. We are currently exploring applying known techniques to these problems [18].

7 Conclusions and future work

This paper presented three key ideas. First, we introduced the idea of XML routers that switch self-describing XML packets based upon any field. Second, we showed how XML routers can be organized into a resilient overlay network that can tolerate both node and link failures without reconfiguration and without interrupting real-time data transport. Finally, we introduced the Diversity Communication Protocol as a way for peers to use redundant packet transmissions to reduce latency and improve reliability.

A wide variety of extensions can be made to the work presently reported, both in protocol refinements and additional functionality. We are actively investigating methods of DCP self-tuning, both for adaptive timers and sophisticated flow control. DCP can also can be used for uninterpreted byte streams. Thus, DCP-like ideas may find application in contexts outside of XML routers. For example, contemporary work on reliable overlay networks (RONs) could use DCP as a RON communication protocol to maximize performance and reliability [2].

Just as secondary storage has become viewed as expendable in pursuit of enhanced functionality and performance [34], we believe that, for certain tightly-constrained applications, network bandwidth across multiple paths may be similarly viewed as well-spent in return for substantial gains in reliability and latency. It is unlikely that multiple disjoint paths with excess capacity will always exist on the last mile to a client. Hence, many installations may benefit from meshes that change to lower levels of redundancy at critical network points such as points-of-presence before last mile cable.

Within the scope of XML routing, our current XML routers could be extended to support.

- More sophisticated XML mesh building and maintenance algorithms.
- Combiners that integrate multiple XML streams for multicast transport as a single stream.
- Using XML routers for duplex communication.

- Other XML network components, such as stream storage and replay.
- Transcoding XML routers that produce output packets that are derivatives of input packets, based upon client queries.

Even in its current form, however, we believe our architecture demonstrates XML is a viable mechanism for content distribution, providing a natural way to encapsulate related data, and a convenient semantic framing mechanism for intelligent network transport and routing.

Acknowledgments

We would like to thank Qian Z. Wang and Micah Gutman for their work on an early version of the XML router and the graphical ASDI client shown in Figure 15. The DCP experiements were conducted at emulab.net, the Utah Network Emulation Testbed, which is primarily supported by NSF grant ANI-00-82493 and Cisco Systems. We are indebted to Benjie Chen and the members of the Click project for assistance with benchmarking our Click-based XML router. This paper greatly benfited from comments on earlier drafts by Chuck Blake, Frans Kaashoek, the anonymous reviewers, and our shepherd, Maurice Herlihy. We also remember Jochen Liedtke, Bruce Jay Nelson, and Mark Weiser as great life forces and friends.

References

[1] XMLBlaster. `http://www.xmlblaster.org/`.

[2] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., AND MORRIS, R. T. Resilient overlay networks. In *Proc. ACM SOSP* (Oct. 2001).

[3] ARMSTRONG, S., ET AL. Multicast transport protocol. RFC 1301, Internet Engineering Task Force, 1992.

[4] BANAVAR, G., CHANDRA, T., MUKHERJEE, B., NAGARAJARAO, J., STROM, R., AND STURMAN, D. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. Int'l Conf. on Dist. Comp. Systems (ICDCS)* (May 1999).

[5] BANERJEA, A. Simulation study of the capacity effects of dispersity routing for fault tolerant realtime channels. In *Proc. ACM SIGCOMM* (Aug. 1996), pp. 194–205.

[6] BESTAVROS, A. An adaptive information dispersal algorithm for time-critical reliable communication. In *Network Management and Control, Volume II*, I. Frish, M. Malek, and S. Panwar, Eds. Plenum Publishing Co., New York, New York, 1994, pp. 423–438.

[7] BRAY, T., ET AL. Extensible markup language 1.0 (second edition). `http://www.w3.org/TR/REC-xml/`, W3C Recommendation, 2000.

[8] BYERS, J. W., LUBY, M., AND MITZENMACHER, M. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *Proc. IEEE Infocom* (Mar. 1999), pp. 275–283.

[9] BYERS, J. W., LUBY, M., MITZENMACHER, M., AND REGE, A. A digital fountain approach to reliable distribution of bulk data. In *Proc. ACM SIGCOMM* (Sept. 1998), pp. 56–67.

[10] CARZANIGA, A., ROSENBLUM, D. S., AND WOLF, A. L. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proc. ACM PODC* (July 2000), pp. 219–227.

[11] CHAMBERLIN, D., ET AL. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`, W3C Working Draft, 2001.

[12] CHAWATHE, Y., MCCANNE, S., AND BREWER, E. RMX: Reliable multicast for heterogeneous networks. In *Proc. IEEE Infocom* (Mar. 2000), pp. 795–804.

[13] CHU, Y., RAO, S. G., AND ZHANG, H. The case for end system multicast. In *Proc. ACM SIGMETRICS* (June 2000), pp. 1–12.

[14] FLOYD, S., JACOBSON, V., MCCANNE, S., LIU, C.-G., AND ZHANG, L. A reliable multicast framework for lightweight sessions and application level framing. *IEEE/ACM Trans. on Networking 5*, 6 (Dec. 1997), 784–803.

[15] HOLBROOK, H. W., AND CHERITON, D. R. IP multicast channels: EXPRESS support for large-scale single-source applications. In *Proc. ACM SIGCOMM* (Aug. 1999), pp. 65–78.

[16] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K., KAASHOEK, M. F., AND O'TOOLE, J. Overcast: Reliable multicasting with an overlay network. In *Proc. USENIX OSDI* (Oct. 2000), pp. 197–212.

[17] KADANSKY, M., CHIU, D., AND WESLEY, J. Tree-based reliable multicast [TRAM]. Technical report TR-98-66, Sun Microsystems Lab, 1998.

[18] KARN, P., AND PARTRIDGE, C. Improving round-trip time estimates in reliable transport protocols. *ACM CCR 17*, 5 (Aug. 1987), 2–7.

[19] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Trans. on Computer Systems 18*, 3 (Aug. 2000), 263–297.

[20] LABOVITZ, C., AHUJA, A., ABOSE, A., AND JAHANIAN, F. Routing stability and convergence. In *Proc. ACM SIGCOMM* (Aug. 2000), pp. 115–126.

[21] LEMPEL, A., AND ZIV, J. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory 23*, 3 (May 1977), 337–343.

[22] LIN, J.-C., AND PAUL, S. RMTP: A reliable multicast transport protocol. In *Proc. IEEE Infocom* (Mar. 1996), pp. 1414–1424.

[23] MANBER, U. Finding similar files in a large file system. In *Proc. Winter USENIX* (Jan. 1994), pp. 1–10.

[24] MAXEMCHUK, N. F. *Dispersity Routing in Store and Forward Networks*. PhD thesis, University of Pennsylvania, May 1975.

[25] MCAULEY, A. J. Reliable broadband communication using a burst erasure correcting code. In *Proc. ACM SIGCOMM* (Sept. 1990), pp. 297–306.

[26] MCCANNE, S., AND JACOBSON, V. Receiver-driven layered multicast. In *Proc. ACM SIGCOMM* (Aug. 1996), pp. 117–130.

[27] MOSER, L., MELLIAR-SMITH, P., AGARWAL, D., BUDHIA, R., AND LINGLEY-PAPADOPOULOS, C. Totem: A fault-tolerant multicast group communication system. *C. ACM 39*, 4 (Apr. 1996), 54–63.

[28] OKI, B., PFLUEGL, M., SIEGEL, A., AND SKEEN, D. The information bus — an architecture for extensible distributed systems. In *Proc. ACM SIGOPS* (Dec. 1993), pp. 58–68.

[29] PAXSON, V. End-to-end internet packet dynamics. *IEEE/ACM Trans. on Networking 7*, 3 (June 1999), 277–292.

[30] PENDARAKIS, D., SHI, S., VERMA, D., AND WALDVOGEL, M. ALMI: An application level multicast infrastructure. In *Proc. USENIX Symp. on Internet Technologies and Systems (USITS)* (Mar. 2001), pp. 49–60.

[31] RABIN, M. O. Efficient dispersal of information for security, load balancing and fault tolerance. *J. ACM 36*, 2 (Apr. 1989), 335–348.

[32] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM CCR 27*, 1 (Jan. 1997).

[33] RIZZO, L., AND VICISANO, L. A reliable multicast data distribution protocol based on software FEC techniques. In *Proc. IEEE HPCS* (June 1997).

[34] SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., AND VEITCH, A. C. Elephant: The file system that never forgets. In *Proc. Workshop on Hot Topics in Operating Systems (HotOS-VII)* (Mar. 1999).

[35] SAVAGE, S., ANDERSON, T., AGGARWAL, A., BECKER, D., CARDWELL, N., COLLINS, A., HOFFMAN, E., SNELL, J., VAHDAT, A., VOELKER, J., AND ZAHORJAN, J. Detour: a case for informed internet routing and transport. *IEEE Micro 19*, 1 (Jan. 1999), 50–59.

[36] SEGALL, B., ARNOLD, D., BOOT, J., HENDERSON, M., AND PHELPS, T. Content based routing with Elvin4. In *Proc. AUUG2K* (June 2000).

[37] STOICA, I., NG, T. S. E., AND ZHANG, H. Reunite: A recursive unicast approach to multicast. In *Proc. IEEE Infocom* (Mar. 2000), pp. 1644–1653.

[38] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Apr. 2000.

[39] TRIDGELL, A., AND MACKERRAS, P. The rsync algorithm. Tech. Rep. TR-CS-96-05, Australian National University, 1997.

[40] VOLPE NATIONAL TRANSPORTATION CENTER, AUTOMATION APPLICATIONS DIVISION. Aircraft situation display to industry functional description and interfaces. DTS-56 report, Aug. 2000.

[41] WHETTEN, B., AND TASKALE, G. An overview of reliable multicast transport protocol II. *IEEE Network 14*, 1 (Jan. 2000), 37–47.

[42] WU, L., SHARMA, R., AND SMITH, B. Thin streams: An architecture for multicasting layered video. In *ProcİEEE Int'l Workshop on Network and Operating System Support for Digital Audio and Video* (May 1997).

[43] YAVATKAR, R., GRIFFIOEN, J., AND SUDAN, M. A reliable dissemination protocol for interactive collaborative applications. In *Proc. ACM Conf. on Multimedia* (Nov. 1995), pp. 371–372.

Appendix H:

Efficient View-Dependent Sampling of Visual Hulls

Wojciech Matusik Chris Buehler Leonard McMillan
Computer Graphics Group
MIT Laboratory for Computer Science
Cambridge, MA 02141

Abstract

In this paper we present an efficient algorithm for sampling visual hulls. Our algorithm computes exact points and normals on the surface of visual hull instead of a more traditional volumetric representation. The main feature that distinguishes our algorithm from previous ones is that it allows for sampling along arbitrary viewing rays with no loss of efficiency. Using this property, we adaptively sample visual hulls to minimize the number of samples needed to attain a given fidelity. In our experiments, the number of samples can typically be reduced by an order of magnitude, resulting in a corresponding performance increase over previous algorithms.

1. Introduction

Recently, shape-from-silhouettes techniques have been used in real-time, shape-acquisition applications [5, 3]. Typically, shape-from-silhouettes techniques involve computing a volume known as the *visual hull*, which is the maximal volume that reproduces the observed silhouettes. It has been found that visual hulls can be computed very quickly, and that the calculation is robust to the crude silhouettes produced by real-time segmentation algorithms.

People commonly compute visual hulls in real-time using one of two approaches: voxel carving [6, 9] and view ray sampling [3]. In voxel carving, a discrete grid of voxels is constructed around the volume of interest. Then, each voxel in the grid is checked against the input sihouettes, and any voxels that project outside the silhouettes are removed from the volume. This procedure results in a view-independent volumetric representation that may contain quantization and aliasing artifacts due to the discrete voxelization. Voxel carving can be accelerated using octree representations.

In view ray sampling, a sampled representation of the visual hull is constructed. The visual hull is sampled in a view-dependent manner: for each viewing ray in some desired view, the intersection points with all surfaces of the visual hull are computed. This procedure removes much of the quantization and aliasing artifacts of voxel carving, but it does not produce a view-independent model. View ray sampling can be accelerated by sampling the visual hull in a regular pattern, such as in a regular grid of pixels.

In this paper, we present a new algorithm for computing view ray sampling. Our algorithm is distinguished by the fact that it allows for computing *arbitrary* patterns of samples with the same efficiency as previous algorithms. We use our algorithm to accelerate visual hull construction by *adaptively* sampling the visual hull. We adjust the density of samples such that more samples are used in regions of large depth variation and fewer samples are used in smooth regions. Using this sampling procedure, we can reduce the number of samples used to construct the visual hull by more than 90% with almost no loss of fidelity. We also demonstrate how to compute surface normals at each sample point. Further, these normals can be used to direct the adaptive sampling procedure.

1.1. Previous Work

Laurentini [2] originated the idea of the *visual hull*: the maximal volume that reproduces all silhouettes of an object. In this paper, visual hulls are constructed from a finite number of silhouettes, so they are only guaranteed to reproduce those particular silhouettes. A visual hull is essentially a volume formed from the intersection of *silhouette cones*. A silhouette cone is the volume that results from extruding a silhouette out from its center of projection. It may be a complicated, non-convex object, depending on the complexity of the silhouette contours. We represent silhouette contours with line segments, so the resulting silhouette cones are facted.

Visual hulls are most often computed using voxel carving[6, 9]. If the primary use for the visual hull is to produce new renderings, then a view ray sampling approach such as the image-based visual hull technique introduced by [3] may be used. The advantage of view ray sampling algorithms is that they do not suffer from the quantization artifacts introduced by discrete volumetric representations.

A series of optimizations are discussed in [3] that reduce the computational overhead of view ray sampling on average to a constant cost per sample. At the heart of these op-

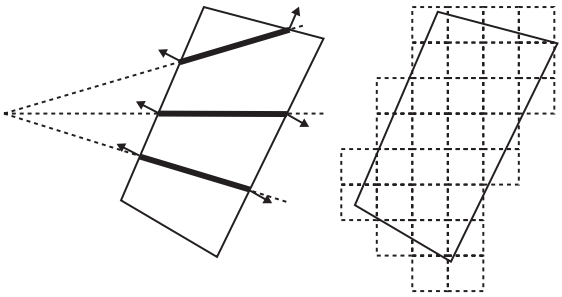


Figure 1: Our algorithms compute a representation of the visual hull that is sampled along viewing rays (left). This is in contrast to a voxel–based representation (right).

timizations is a presorting of the silhouette edges about the epipole of the desired image. This sorting allows samples to be computed in constant time if the samples are arranged in scanline order. This initial sorting also involves some additional cost.

1.2. Contributions

In this paper, we present an improved algorithm that allows the samples to be computed efficiently in any order. This flexibility makes it easy to do hierarchical or adaptive sampling of the visual hull. In addition this algorithm visits the silhouettes in a "lazy" fashion, using only the portions of the silhouettes necessary. We also describe how to compute normals for the visual hull samples, which can be useful for shading and surface reconstruction.

2. Visual Hull Sampling

One way to compute an exact sampling of a visual hull consists of two steps: (1) compute a set of polygons that defines the surface of a visual hull and (2) sample these polygons along the sampling rays to produce exact visual hull samples. The first step consists of a 3D intersection of all the extruded silhouette cones. The second step is also an intersection operation–an intersection of sampling rays with the volume. The ray intersections result in a set of occupied volume intervals along each sampling ray.

The algorithm in section 2 is conceptually simple. However, it is impractical to compute full 3D intersections between silhouette cones, especially when we are ultimately interested in samples. We would like to compute the same result without performing full volume–volume intersections. To do this, we take advantage of the commutative properties of the intersection operation. An intersection of a ray with a visual hull is described mathematically as follows:

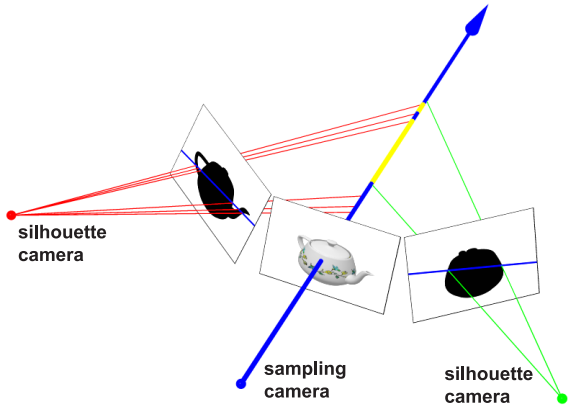


Figure 2: Instead of computing the intersection of the sampling ray with extruded silhouette cones, we can project the sampling ray onto the silhouette images, compute the intersections of the projected sampling rays with the silhouette, and lift back the resulting intervals to 3D.

$$VH(S) = \left(\bigcap_{s \epsilon S} cone(s) \right) \cap ray3D \qquad (1)$$

This operation is equivalent to:

$$VH(S) = \bigcap_{s \epsilon S} (cone(s) \cap ray3D) \qquad (2)$$

This means that we can first intersect each extruded silhouette with the 3D ray separately. This results in a set of occupied intervals along the ray. Then we compute the intersection of all these sets of intervals for all silhouettes. In this process we exchanged volume-volume and volume-line intersections for simpler volume-line and line-line intersections.

2.1. Image Space Intersections

We can further simplify the intersection process by exploiting the fact that the cross–section of the extruded silhouette remains fixed. This observation implies that instead of computing the ray intersection with the *extruded* silhouette, we can compute the intersection of the silhouette with the ray projected onto the plane that defines the cross-section. We can then backproject the results of the image space intersection onto the original 3D ray (see Figure 2). Effectively we reduce the volume-line intersections to area-line intersections. As we will see in the next section, this reduction allows us to use simple 2D spatial partitioning for accelerating ray intersections.

We can pick any plane that completely intersects the silhouette cone when we perform the area-line intersection.

However, it is simplest to perform the operation on the silhouette camera's image plane because (1) the cross-section is already defined for this plane (it is the silhouette) and (2) we avoid any possible resampling artifacts.

3. Silhouette-Line Intersections

In this section, we describe an efficient technique for computing the intersection of a set of 2D lines with silhouette contours. We impose one constraint on the sampling rays: we require that they all emanate from a single point in space, the *sampling camera*. This limitation on our algorithm is not too severe, as it is often desired to sample the visual hull from the point-of-view of an arbitrary camera or one of the silhouette cameras. Note that we do not constrain the sampling rays to be on a regular grid or in any other structured organization. Also, the algorithm can work with a set of parallel sampling rays. This case corresponds to an orthographic sampling camera with a center of projection at infinity.

Our algorithm is based on a data structure called a "Wedge-Cache." The Wedge-Cache is used to store intersection information that can be re-used when computing later intersections. The main idea behind the Wedge-Cache algorithm is based on the epipolar geometry of two views. An epipolar plane (i.e., a plane that passes through centers of projections of both cameras) intersects the image planes of both cameras in epipolar lines [1]. It is easy to see that all rays from the first camera that lie in the epipolar plane project to the exact same epipolar line in the second camera (of course, the reverse is true too). Therefore, many sampling rays will project to the same line in any given silhouette view. The Wedge-Cache algorithm takes advantage of this fact to compute the fast line-silhouette intersections. The basic idea is to compute and store the intersection results for each epipolar line the *first* time it is encountered. Then, when the same epipolar line is encountered again, we can simply look up previously computed results.

In practice (because of discretization), we rarely encounter the exact same epipolar line twice. Since we want to reuse silhouette intersections with epipolar lines, we compute and store the intersections of the silhouette with wedges (i.e., sets of epipolar lines) rather than single lines. Within each wedge, we store a list of silhouette edges that individual lines within the wedge might intersect. Then, when an epipolar line falls within a previously computed wedge, we need only intersect that line with the silhouette edges belonging to the wedge.

We discretize the silhouette image space into a set of wedges such that each wedge has exactly one pixel width at the image boundaries (see Figure 3). Depending on the position of the epipole (shown as a dot) with respect to the silhouette image, we distinguish nine possible cases of image boundary parts that need to be used. These cases are illustrated in Figure 4. There is only one special case that needs to be handled: the case in which the epipolar lines in a silhouette camera are parallel and do not converge at the epipole (i.e., the epipole is at infinity). In this case, we can modify the Wedge-Cache to use parallelogram-shaped "wedges" (see Figure 5). In some applications, this case can be avoided by a small perturbation in the orientation of the sampling camera.

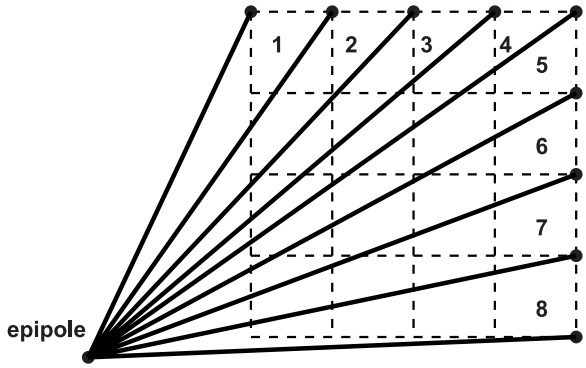


Figure 3: The silhouette image is partitioned into a set of wedges, which are the entries of the Wedge-Cache. In this simple example, the silhouette image is 4×4 pixels and there are eight Wedge-Cache entries.

Execution of the Wedge-Cache algorithm proceeds as follows. For each sampling ray we compute its epipolar line. Then we determine into which wedge it falls. If silhouette edges that lie in this wedge have not been computed, we use a Bresenham-like algorithm to traverse all the pixels in the wedge and compute these edges. Then, we test which of the computed edges in the wedge actually intersect the given epipolar line. Later, when other epipolar lines fall into this wedge we simply look up the edges contained in the wedge and test for intersection with the epipolar line. Figure 6 illustrates a simple Wedge-Cache with two wedges. The silhouette in this case is a square consisting of four edges $a, b, c,$ and d. Each wedge contains three edges as shown in the figure. In this example, the epipolar line corresponding to some sampling ray is contained in wedge 1. Checking against the three edges in wedge 1 reveals that the line intersects two edges, a and d.

One nice property of the Wedge-Cache algorithm is that it employs a lazy-computation strategy; we process pixels and edges in the wedge only when we have an epipolar line that lies in this wedge. The pseudocode for `VHsample` that uses the Wedge-Cache Algorithm is shown in Figure 7.

```
VHsample (samplingRays R, silhouettes S)
  for each silhouetteImage s in S
    compute_silhouette_edges(s)
    for each samplingRay s in R do
      r.intervals = 0..inf
    for each silhouetteImage s in S
      clear(Cache)
      for each samplingRay r in R
        lineSegment2D l2 = project_3D_ray(r,s.camInfo)
        integer index = compute_wedge_cache_index(l2)
        if Cache[index] == EMPTY
          silhouetteEdges E = trace_epipolar_wedge(index, s)
          Cache[index] = E
        intervals int2D = linesegment_isect_silhouette(l2,Cache[index])
        intervals int3D = lift_up_to_3D(int2D,r.camInfo,ry3)
        r.intervals = intervals_isect_intervals(r.intervals,int3D)
```

Figure 7: Pseudocode for Wedge-Cache algorithm.

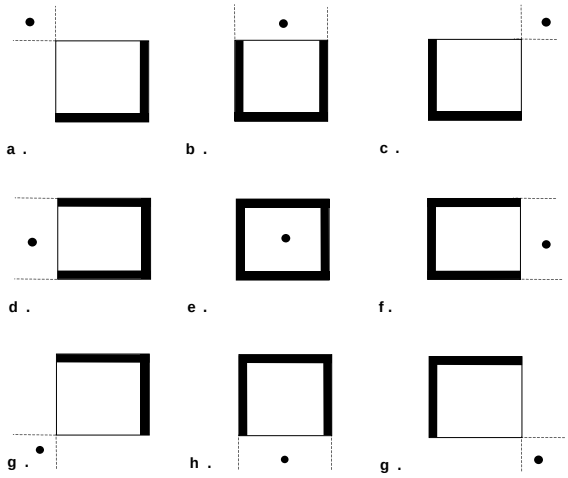


Figure 4: Depending on the position of the epipole with respect to the silhouette image boundary, we decide which parts of the silhouette image boundary (thick black lines) need to be used for wedge indexing.

3.1. Using Valid Epipolar Line Segments

Some care must be taken when implementing the `calc2Dintervals` subroutine. This is because some portions of the epipolar line are not valid and should not be considered. There are two constraints on the extent of the epipolar line: (1) it must be in front of the sampling camera and (2) it must be seen by the silhouette camera. The two constraints result in four cases that are easily distinguished; see [4] for details.

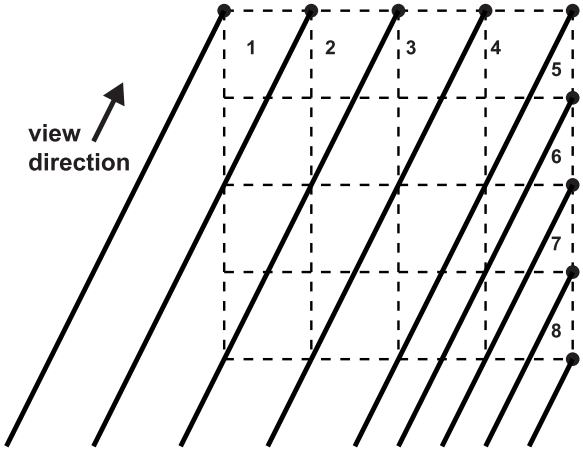


Figure 5: In the special case when the epipolar lines are all parallel, the wedges become parallelograms.

3.2. Visual Hull Surface Normals

In this section we show how to compute visual hull surface normals for each of the interval endpoints of the sampled visual hull representation. The surface normal is useful to reconstruct the surface of the visual hull, and we use normals in this manner to control the adaptive sampling procedure described in Section 4. Of course, the normal of the visual hull is not the same as the normal of the original object. However, as the number of silhouettes increases, the normals of the visual hull approach the normals of the object in non-concave regions.

Normals are simple to compute with just a little extra bookkeeping. For each interval endpoint we store a refer-

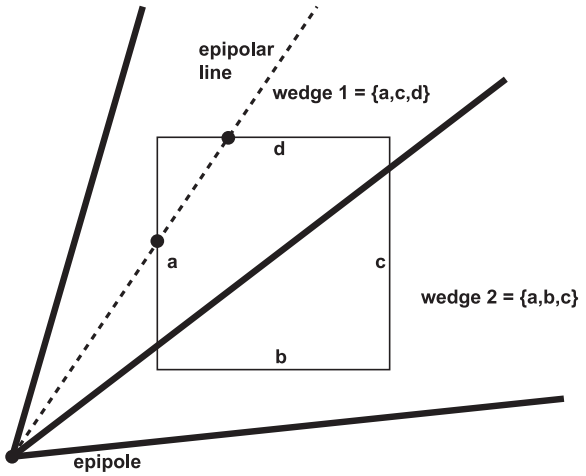


Figure 6: A simple Wedge-Cache example with a square silhouette. The epipolar line is contained in wedge 1, so it need only be compared to edges a, c, and d for intersection. In fact, the line intersects only edges a and d.

ence to the silhouette edge and the silhouette image that determine the interval endpoint. Each interval is then defined as $((depth_{start}, edge_{i,m}), (depth_{end}, edge_{j,n}))$, where i and j are the indices of the reference images and m and n are the silhouette edge indices in the reference images. The stored edges and the center of projection of the corresponding reference image determine a plane in 3D. The normal of this plane is the same as the surface normal of the point on the visual hull (see Figure 8). We can compute the plane normal using the cross-product of the two vectors on this plane. This leaves two choices of normals (differing in sign); the proper normal can be chosen based on the direction of the sampling ray and the knowledge of whether the sampling ray is entering or leaving the visual hull.

4 Adaptive Visual Hull Sampling

One advantage of the Wedge Cache algorithm is that it allows for sampling along arbitrary viewing rays in any order. We have used this property to implement an adaptive sampling procedure that can drastically reduce the number of samples required to construct an image of the visual hull from a particular view.

First, we decide upon a minimum size N of features that we expect to see in the visual hull. This choice determines the smallest features of the visual hull that our algorithm can resolve. Typically, we choose a minimum feature size of $N = 4$, which means the algorithm can potentially miss features smaller than 4×4 pixels.

Next, we perform an initial sampling of the visual hull

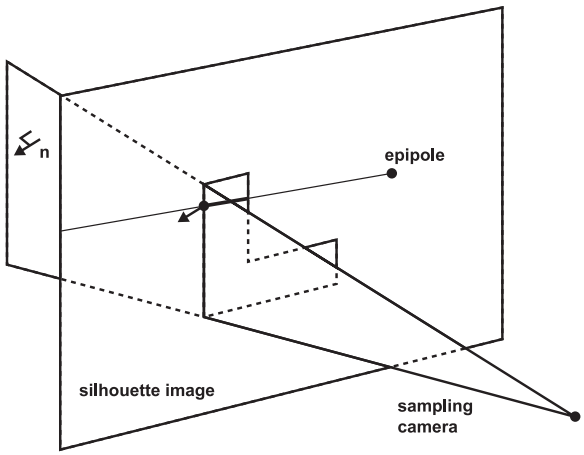


Figure 8: A silhouette edge combined with the center of projection define a plane. Visual hull interval endpoints defined by that edge have the same normal as the plane.

by sampling every N^{th} pixel in both the x and y directions. This initial sampling results in a coarse grid of samples over the image. For each square in the grid, we consider the samples at its four corners. If all four sample rays miss the visual hull, then we conclude that no rays within the square hit the visual hull, and we mark that square as empty. If some of the rays miss the visual hull and some of them do not, then we conclude that a silhouette boundary passes through this square. In this case, we sample the rest of the rays within the square to resolve the silhouette edge accurately.

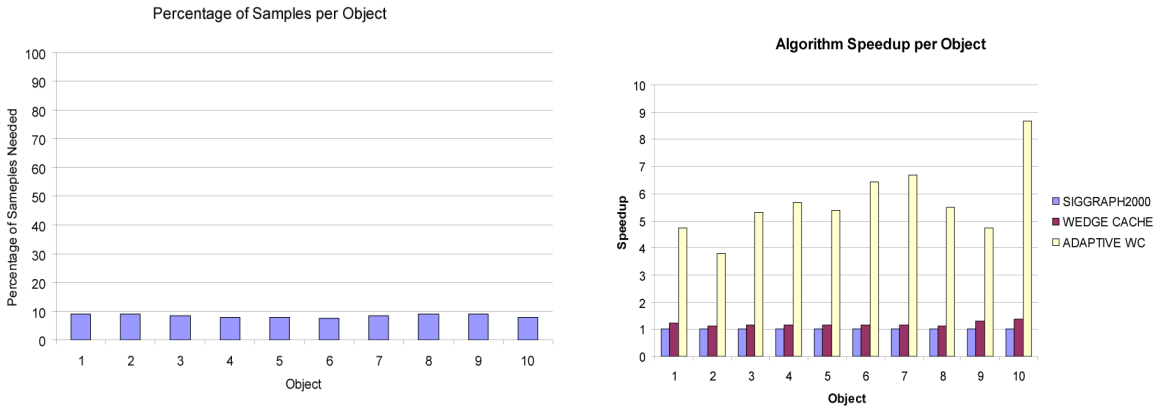
If all four sample rays hit the visual hull, then we decide whether to sample further based on a simple estimate of the surface continuity. Since we know the normals at each of the four samples, we can construct a plane through one the sample points on the visual hull. If this plane does not predict the other three sample points sufficiently well, then we sample the rest of the rays within the square. Otherwise, we approximate the depths of the samples within the square using the planar estimate. We compare the prediction error against a global threshold to decide if more sampling is necessary.

5. Results

In this section, we present some results of our adaptive sampling algorithm. In Figure 9a, we show the depth map of a visual hull sampled at every pixel. This visual hull is computed from 108 input images, and it is sampled at 1024×1024 resolution (1048576 samples). In Figure 9b, we have the same view of the visual hull, but this time it is sampled with only about 9% of the samples. The mean squared error between the two depth maps is 0.1 (the depth maps are quantized to 8 bits). Figure 9c shows the sampling

(a) (b) (c)

Figure 9: Visual hull sampling results. (a) Shows the result of sampling every pixel. (b) Shows the result of adaptively sampling only about 9% of the pixels. (c) Shows where the adaptive sampling procedure increased the sampling density.



(a) (b)

Figure 10: Wedge cache performance results. (a) Shows the percentage of samples that are computed using adaptive sampling. (b) Shows wedge cache performance speedup as compared to the algorithm in [3].

pattern used for this view of the visual hull.

We have found that adaptive sampling of visual hulls is fruitful on a wide variety of different objects. In Figure 10a we have plotted the percentage of samples needed to render a visual hull given a fixed threshold for evaluating surface continuity.

We compared the runtime performance of our algorithm to the algorithm described in [3]. The results are shown in Figure 10b. When fully sampling every pixel in the image, our wedge cache algorithm is slightly faster on all objects. When we enable adaptive sampling, the speedup is more dramatic. On average, there is a factor of five speedup, which would directly result in increased framerates.

6. Conclusions

We have presented a new algorithm for efficiently sampling visual hulls. Our algorithm computes exact samples of the visual hull surface along with surface normals. The algorithm computes these samples along an arbitrary set of sampling rays that emanate from a common point, possibly at infinity.

Using this algorithm, we have demonstrated a simple way to adaptively sample depth maps of visual hulls from virtual viewpoints. By using this adaptive sampling, the performance of the algorithm is increased on average 5 times with almost no loss in quality of the resulting depth maps. In real-time systems, for which visual hull analysis has been found useful, this performance increase translates directly into faster framerates.

References

[1] Faugeras, O. *Three-Dimensional Computer Vision.* MIT Press. 1993.

[2] Laurentini, A. “The Visual Hull Concept for Silhouette Based Image Understanding.” *IEEE PAMI* 16,2 (1994), 150-162.

[3] Matusik, W., Buehler, C., Raskar, R., Gortler, S., and McMillan, L. “Image-Based Visual Hulls,” *SIGGRAPH 2000*, July 23-28, 2000, 369-374.

[4] McMillan, L. “An Image-Based Approach to Three-Dimensional Computer Graphics,” Ph.D. Thesis, University of North Carolina at Chapel Hill, Dept. of Computer Science, 1997.

[5] Kanade, T., P. W. Rander, P. J. Narayanan. “Virtualized Reality: Constructing Virtual Worlds from Real Scenes,” *IEEE Multimedia*, 4, 1 (March 1997), pp. 34-47.

[6] Potmesil, M. ”Generating Octree Models of 3D Objects from their Silhouettes in a Sequence of Images.” *CVGIP 40* (1987), 1-29.

[7] Roth, S. D., “Ray Casting for Modeling Solids.” *Computer Graphics and Image Processsing*, 18 (February 1982), 109-144.

[8] Snow, D., Viola, P., and Zabih, R., “Exact Voxel Occupancy with Graph Cuts,” *Proceedings IEEE Conf. on Computer Vision and Pattern Recognition.* 2000.

[9] Szeliski, R. “Rapid Octree Construction from Image Sequences.” *CVGIP: Image Understanding* 58, 1 (July 1993), 23-32.